Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Физико-технологический институт

И. Н. ОГОРОДНИКОВ

МИКРОПРОЦЕССОРНАЯ ТЕХНИКА. ВВЕДЕНИЕ В KEIL C51

Учебное пособие

Министерство науки и высшего образования
Российской Федерации

Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

И. Н. Огородников

МИКРОПРОЦЕССОРНАЯ ТЕХНИКА. ВВЕДЕНИЕ В KEIL C51

Учебное пособие

Рекомендовано методическим советом
Уральского федерального университета для студентов вуза,
обучающихся по направлениям подготовки
14.03.02 «Ядерные физика и технологии»,
12.03.04 «Биотехнические системы и технологии»,
14.05.04 «Электроника и автоматика физических установок»

Екатеринбург
Издательство Уральского университета
2021

УДК 004.31:004.42(075.8)
ББК 32.973.26-04я73
О-39

Рецензенты:
кафедра высшей математики и физики Уральского технического института связи и информатики СибГУТИ (завкафедрой высшей математики и физики, канд. физ.-мат. наук, доц. *В. Т. Куанышев*);
директор ООО «Инжетех» *С. В. Богушевич*

Научный редактор — канд. физ.-мат. наук, доц. *И. Н. Анцыгин*

Для оформления обложки использовано изображение из личного архива автора.

Огородников, И. Н.

О-39 Микропроцессорная техника. Введение в Keil C51 : учебное пособие / И. Н. Огородников ; М-во науки и высшего образования РФ. — Екатеринбург : Изд-во Урал. ун-та, 2021. — 100 с.

ISBN 978-5-7996-3301-1

Учебное пособие нацелено на формирование у студентов практических навыков разработки и программирования микропроцессорных устройств автоматики физических установок, приборов радиационной безопасности человека и окружающей среды, а также различных приборов биофизического и медицинского назначения. Предназначено для студентов технических специальностей Физико-технологического института Уральского федерального университета всех уровней обучения.

Библиогр.: 11 назв. Табл. 20. Рис. 28.

УДК 004.31:004.42(075.8)
ББК 32.973.26-04я73

ISBN 978-5-7996-3301-1

Содержание

Предисловие

Представляемое учебное пособие подготовлено на практическом материале, накопленном автором при преподавании в Уральском федеральном университете двухсеместрового курса по основам микропроцессорной техники, который изучается студентами Физико-технологического института УрФУ, специализирующимися в областях электроники и автоматики физических установок, приборов для применения в области радиационной безопасности человека и окружающей среды, защиты от излучений, радиационной экологии, биомедицинской инженерии. Пособие может быть полезно студентам других родственных специальностей. Рассмотрены практические вопросы программирования микропроцессорных устройств, а также оформления программ, разработанных при проектировании микропроцессорных устройств. Данное учебное пособие касается лишь практической части учебного курса по основам микропроцессорной техники. Оно содержит дополнительный теоретический материал и техническую информацию, которые необходимы студентам при программировании микропроцессорных устройств на платформе `x51` во время практических занятий, лабораторного практикума, а также при выполнении индивидуального домашнего задания и курсовой работы. Отсюда название — введение в Keil C51. При подготовке учебного пособия и подборе материалов к нему автор полагал, что читатель уже знаком (в объеме программы технического вуза) с основами следующих дисциплин: теория цепей и сигналов, физические основы и элементная база электронной техники, аналоговая схемотехника, цифровая и импульсная техника, основы микропроцессорной техники, программирование на языках высокого и низкого уровней. Для углубленного восприятия излагаемого материала желательно предварительное знакомство читателя с теоретическими и практическими основами микропроцессорной техники, краткое изложение которых содержится в ранее изданных автором учебнике [1] и учебном пособии [2]; основами программирования микроконтроллеров семейства `x51` на языках ассемблера [3] и `C51` [4]; возможностями инструментальных средств для платформы `x51`: руководство [5], а также прил. А.

Следует отметить, что микропроцессорная техника представляет собой обширную, динамично развивающуюся область знаний. В пособие включен лишь ограниченный круг вопросов, выбор и глубина освещения которых продиктованы требованиями федеральных государственных образовательных стандартов высшего образования по направлениям подготовки 14.03.02 «Ядерные физика и технологии», 12.03.02 «Биотехнические системы и технологии» и специальности 14.05.04 «Электроника и автоматика физических установок».

В первой главе приведены необходимые сведения о синтаксисе языка программирования Keil C51. После общей характеристики и описания синтаксиса, используемых символов, ключевых слов и идентификаторов обсуждаются форматы данных и подробно раскрывается значение наиболее важных ключевых слов, включая директивы и модификаторы памяти. Далее идет обсуждение операторов языка программирования и выражения. Уделено внимание понятию указателей, используемых в архитектуре микроконтроллера `x51`: нетипизированные (*generic pointer*) и память-зависимые (*memory-specific*) указатели. Специальное внимание уделено обсуждению различий в объявлениях этих указателей, форматы, получаемый ассемблерный код, быстродействие фрагментов кода, в котором применяются данные указатели, явный и неявный способы преобразования типов указателей. В завершении главы рассматриваются механизмы передачи параметров и возвращения данных при взаимодействии программ и функций.

Вторая глава посвящена особенностям взаимодействия программных модулей, написанных на языка ассемблера (A-программа) и Keil C51 (C-программа): соглашение по взаимодействию, настройка стартового адреса, обращение к регистрам, программирование прерываний, вызову A-программ из C-программ. Кратко обсуждаются стандартные библиотеки компилятора Keil C51.

Третья глава посвящена разработке и описанию авторской библиотеки учебных программ `sdk_base`, предназначенных для доступа к оборудованию учебного стенда: регистрам `ПЛИС`, параллельным и последовательным (`UART` и `I2C`) портам, дисплею, клавиатуре и т. п. Приведены примеры вызова различных библиотечных процедур (A-функция) из `C`-программы. Обсуждаются организация и исходные тексты библиотечных модулей, краткая сводка которых дана в прил. Б.

В четвертой главе сформулированы общие требования к оформлению учебных C-программ: соглашения по идентификаторам, самодокументированности и читаемости C-программ.

Для закрепления теоретического материала в рамках учебной дисциплины предусмотрена курсовая работа. В прил. В и Г обсуждаются вопросы организации курсовой работы и оформления отчета: общие положения, нормативные документы, цели, задачи и тематика работ, порядок выполнения, общие требования к составу работы и оформлению отчета, этапы контроля и отчетности по курсовой работе.

Отметим, что рассматриваемый двухсеместровый курс основ микропроцессорной техники является вводным курсом перед последующим изучением более сложного профессионально-ориентированного курса «Микропроцессорные системы».

1. Синтаксис Keil C51

Keil C51 – это язык программирования высокого уровня, предназначенный для применения в качестве инструмента структурно-модульного программирования платформы `x51`. В общем случае в состав любой программы, написанной на языке `C51`, могут входить несколько различных модулей. Наличие в программе хотя бы одного модуля является обязательным, поэтому в крайнем случае программа на `C51` должна состоять как минимум из одного модуля. Исходный текст каждого модуля при этом компилируется отдельно и размещается в отдельном файле.

Структура любой программы, созданной компилятором `x51`, состоит из нескольких подпрограмм, одна из которых получает обязательное имя `main`, содержит одноименную точку входа и называется далее *основной программой*. Выполнение любой программы, созданной компилятором Keil C51, всегда начинается с основной программы `main`, которая может вызывать другие подпрограммы, размещенные в различных модулях. Программный проект может содержать несколько объектных модулей, сохраненных в разных файлах и написанных, возможно, на разных языках программирования, но удовлетворяющих соглашения о передаче параметров и возвращаемых значениях, принятых в Keil C51. При разработке любых программ для компилятора Keil C51 необходимо обязательно учитывать требования синтаксиса этого языка, которые рассматриваются ниже.

1.1. Символы, ключевые слова и идентификаторы

При назначении ключевых слов и идентификаторов используются буквы английского алфавита, цифры и символ подчеркивания '_'. Отметим, что компилятор различает прописные и строчные буквы. Так, например, `bytes` и `Bytes` в языке `C51` являются двумя различными идентификаторами. Для создания операторов языка и организации вычислительного процесса в Keil C51, так же, как и в стандартном ANSI C, используются специальные символы (табл. 1.1).

Помимо ключевых слов и идентификаторов, исходный текст программы содержит также *разделительные* и *управляющие* символы. К разделительным символам относятся пробел, символы табуляции, перевода строки, возврата каретки, символы новой страницы и новой строки. Все они служат для разделения лексических единиц языка программирования, таких как ключевые слова, константы, идентифи-

Таблица 1.1

Специальные символы языка Keil C51

Символ	Наименование	Символ	Наименование
,	Запятая	)	Круглая скобка правая
.	Точка	(	Круглая скобка левая
;	Точка с запятой	}	Фигурная скобка правая
:	Двоеточие	{	Фигурная скобка левая
?	Вопросительный знак	<	Меньше
'	Апостроф	>	Больше
!	Восклицательный знак	[	Квадратная скобка правая
|	Вертикальная черта	]	Квадратная скобка левая
/	Дробная черта	#	Октоторп (решетка)
\	Обратная черта	%	Процент
~	Тильда	&	Амперсанд
*	Астериск (звездочка)	^	Исключающее ИЛИ
+	Плюс	=	Равно
–	Минус	"	Кавычки

каторы и т. п. Если в тексте программы встречается подряд несколько одинаковых разделительных символов (например, последовательность пробелов или символов табуляции), то компилятор интерпретирует их как один разделительный символ.

Особую группу символов составляют управляющие последовательности языка Keil C51, которые представляют собой специальные символьные комбинации, используемые в функциях ввода-вывода данных. Управляющая последовательность всегда начинается с символа «обратный слеш» '\' (англ. *backslash*), за которым следует комбинация латинских букв и цифр (табл. 1.2).

Так, любой символ из кодовой таблицы ASCII или ANSI можно представить в виде последовательности восьмеричных или шестнадцатеричных цифр, используя управляющие последовательности `\OOO` или `\xHHH` (табл. 1.2). Например, символ возврата каретки, для которого определена индивидуальная управляющая последовательность `\r`, можно представить также через восьмеричный (`\015`) или шестнадцатеричный (`\x00D`) коды.

В табл. 1.2 приведен исчерпывающий список управляющих последовательностей компилятора C51. Если попытаться записать произвольную последовательность, состоящую из обратного слеша и какого-либо символа, который не перечислен в табл. 1.2 и не является цифрой, то компилятор попросту проигнорирует знак обратного слеша, а следующий за ним символ будет трактовать как литеральный (т. е. одиночный) символ.

Таблица 1.2

Управляющие последовательности языка Keil C51

Управляющая последовательность	Наименование	Шестнадцатеричный код
\a	Звонок	007
\b	Возврат на шаг	008
\t	Горизонтальная табуляция	009
\n	Переход на новую строку	00A
\v	Вертикальная табуляция	00B
\r	Возврат каретки	00D
\f	Новая страница	00C
\"	Кавычки	022
\'	Апостроф	027
\0	Нуль-символ	000
\\	Обратная дробная черта	05C
\000	8-ричный код ASCII- или ANSI-символа	—
\xHHH	16-ричный код ASCII- или ANSI-символа	HHH

Для примера составим последовательность '\c'. Компилятор будет воспринимать ее как одиночный символ 'c'. Если эту последовательность указать в строке среди других символов, то компилятор воспримет ее как символ 'c' в составе строковой или символьной константы, т. е. `"Ab\cdef" = "Abcdef"`.

На практике обратный слеш можно использовать в качестве символа продолжения, когда длинную строку необходимо разбить на две короткие строки. Если за символом '\' следует управляющий символ возврата каретки, то оба эти символа игнорируются и компилятор считает следующую строку продолжением предыдущей.

Для определения имени переменной, подпрограммы, символической константы или метки оператора в языке Keil C51 используются идентификаторы, длина которых может достигать 255 символов. Однако при распознавании идентификаторов компилятор различает только первые 31 символ и игнорирует все остальные символы.

Идентификатор состоит из последовательности символов, в которую могут входить любые прописные или строчные буквы латинского алфавита, символ подчеркивания '_', цифры. При определении идентификатора следует учитывать два ограничения: 1) первым символом идентификатора должна быть буква или символ подчеркивания '_', но не цифра; 2) компилятор Keil C51 различает регистр букв.

Непосредственное создание идентификатора происходит при объявлении переменной, функции, структуры, объединения и т. п. Идентификатор не должен совпадать ни с ключевыми словами, ни с за-

резервированными словами, ни с именами функций из библиотеки компилятора языка Keil C51. Отметим, что символ подчеркивания '_' широко используется в именах системных функций и переменных, поэтому настоятельно не рекомендуется использовать его в качестве первого символа в создаваемых пользователем идентификаторах, поскольку такой идентификатор может совпасть с именем какой-либо системной функции или переменной, которая в результате этого станет недоступной.

В языке программирования Keil C51 определены те же самые ключевые слова, которые применяют в стандартном ANSI C. Однако в дополнение к этому в Keil C51 используется целый ряд новых ключевых слов, используемых только при программировании платформы **x51**, которых в большинстве своем нет в стандартном ANSI C (табл. 1.3).

Таблица 1.3

Ключевые слова Keil C51 для платформы x51

№ п/п	Слово	№ п/п	Слово	№ п/п	Слово	№ п/п	Слово
1.	`_at_`	6.	`compact`	11.	`pdata`	16.	`sfr16`
2.	`alien`	7.	`data`	12.	`_priority_`	17.	`small`
3.	`bdata`	8.	`far`	13.	`reentrant`	18.	`_task_`
4.	`bit`	9.	`idata`	14.	`sbit`	19.	`using`
5.	`code`	10.	`interrupt`	15.	`sfr`	20.	`xdata`

Идентификаторы пользователя не должны совпадать с ключевыми словами. Далее будет проведено подробное обсуждение смысла каждого из ключевых слов, перечисленных в табл. 1.3.

В языке Keil C51 предусмотрено определение *констант*, которые так же, как и в классическом ANSI C, предназначены для введения чисел в состав выражений операторов языка программирования C51. Константы всегда начинаются с цифры в отличие от идентификаторов, которые всегда начинаются с буквы. В языке C51 можно использовать следующие типы констант:

- целые знаковые и беззнаковые константы;
- константы с плавающей точкой;
- символьные константы и литеральные строки.

Для записи целочисленных констант используются восьмеричная, десятичная или шестнадцатеричная формы представления. При этом десятичная константа, состоящая из одной или нескольких десятичных цифр, не может начинаться с нуля, иначе компилятор интерпретирует это число как восьмеричное.

Восьмеричная константа всегда начинается с обязательного нуля. За ним могут следовать одна или несколько восьмеричных цифр, каждая из которых лежит в диапазоне от 0 до 7. Шестнадцатеричная константа всегда начинается с обязательного префикса – после-

довательности символов `0x` или `0X`. После префикса следуют одна или несколько шестнадцатеричных цифр, каждая из которых лежит в диапазоне от `0` до `F`. Приведем примеры записи беззнаковых (англ. *unsigned*) целочисленных констант: `11`, `127` (десятичные константы); `013`, `077` (восьмеричные константы); `0x2A`, `0x1F` (шестнадцатеричнык константы). Знаковая (англ. *signed*) целочисленная константа может иметь положительное значение (знак «плюс» перед константой обычно опускают) или отрицательное значение (знак «минус» предшествует константе), например, `-14`, `-0x2A`, `-057`.

При объявлении любой целочисленной константы ей обязательно присваивается тип, который задает необходимые преобразования, чтобы константу можно было использовать в выражениях. Так, десятичные константы трактуются как знаковые числа, которым присваивается тип `int` (целое число) или `long` (длинное целое число) в зависимости от численного значения этой константы. Пороговой величиной является число 2^{15} (32768): если значение константы меньше пороговой величины, то ей присваивается тип `int`, в противном случае константе присваивается тип `long`. Восьмеричные и шестнадцатеричные константы, в зависимости от своей численной величины, могут быть преобразованы к типу `int`, `uint`, `long` или `unsigned long`.

Для представления константы с плавающей точкой (англ. *floating point*, FP) используется стандартная экспоненциальная форма записи, состоящая из мантиссы (англ. *mantissa*) и порядка (англ. *exponent*) числа:

$$N = M \times 10^P,$$

где N – представляемое число; M – мантисса, т. е. дробное знаковое действительное число с десятичной точкой ($1 \leqslant M < 10$); P – порядок, т. е. целое знаковое число.

Компьютерный способ экспоненциальной записи вместо основания показательной функции (10) использует букву `E` (англ. *exponent*), следом за которой записывается показатель P. Компьютерный формат константы с плавающей точкой имеет следующий общий вид:

[цифры].[цифры] [E|e [+|−] цифры].

Приведем несколько примеров записи констант с плавающей точкой: `75.19`, `1.1E-3`, `-0.003`, `.015`, `-0.32e2`.

Запись символьной константы состоит из ASCII- или ANSI-символа, заключенного в апострофы. Отметим, что управляющие последовательности также могут быть представлены в виде символьных констант, однако они при этом рассматриваются как один символ. Приведем несколько примеров символьных констант:

`' '` – пробел;

`'S'` – буква S;

`'\n'` – символ новой строки;

'\\' – обратная дробная черта;

'\v' – вертикальная табуляция.

Символьным константам присваивается тип `int`, а при преобразовании типов константы могут дополняться каким-либо знаком.

Для отображения сообщений в языке Keil C51 предусмотрены строковые константы, или, по-другому, строковые *литералы* (англ. *literal*), которые представляет собой последовательности любых символов (буквы и цифры), заключенных в кавычки (`"..."`). Приведем примеры строковых констант: `"Character input"`, `"Output value"`. В строковых константах допустимо использовать пробелы.

Следует отметить, что для представления в литеральной строке управляющих символов, таких как кавычка '"', обратный слеш '\' и символ новой строки '\n', необходимо использовать соответствующие управляющие последовательности.

Символы литеральной строки могут размещаться в памяти программ или памяти данных. Компилятор добавляет дополнительный нуль-символ '\0' в конец каждой литеральной строки. Нуль-символ указывает на завершение строки, поскольку компилятор Keil C51, как и для случая классического ANSI C, выполняет операции над строками, завершающимися обязательным нуль-символом (англ. *null-terminated string*).

При обработке литеральной строки компилятор интерпретирует ее как символьный массив. При этом общее количество элементов символьного массива равно числу символов в строке плюс единица, т. к. учитывается завершающий символ '\0'.

1.2. Форматы данных

В языке программирования Keil C51 предусмотрены различные типы данных, с которыми может работать компилятор. В табл. 1.4 приведена сводная информация об этих типах данных и их форматах.

Битовые данные (`bit`) принимают значения 0 или 1, недоступны через указатели, т. к. специфичны для C51 и не входят в ANSI C.

Однобайтовые символы: со знаком (`signed char`) и без знака (`unsigned char` или `uchar`). Можно использовать общепринятое сокращение и писать `uchar` вместо `unsigned char`.

Двухбайтовые целые: со знаком (`signed int`, `signed short`) и без знака (`unsigned int` или `uint`, `unsigned short`). Можно использовать общепринятое сокращение и писать `uint` вместо `unsigned int`. В языке Keil C51 тип *короткие целые* (`short`) по формату не отличается от типа данных *целые* (`int`).

Четырехбайтовые длинные целые со знаком (`signed long`) и без знака (`unsigned long`). В этом случае сокращения не используют.

Таблица 1.4

Типы данных Keil C51

Тип данных	Формат, бит	Формат, байт	Диапазон значений
Бит (`bit`)[1]	1		0 или 1
Символ со знаком (`signed char`)	8	1	−128...+127
Символ без знака (`uchar`)[2]	8	1	0...255
Короткое целое со знаком (`signed short`)	16	2	−32768...+32767
Короткое целое без знака (`unsigned short`)	16	2	0...65535
Целое со знаком (`signed int`)	16	2	−32768...+32767
Целое без знака (`uint`)[2]	16	2	0...65535
Длинное целое со знаком (`signed long`)	32	4	−2147483648...+2147483647
Длинное целое без знака (`unsigned long`)	32	4	0...4294967295
Число с плавающей точкой (`float`)	32	4	±1.175494E−38...±3.402823E+38
Указатели `data*`, `idata*`, `pdata*`	8	1	0x00...0xFF
Указатели `code*`, `xdata*`	16	2	0x0000...0xFFFF
Указатель нетипизированный (`generic pointer`)	24	3	Тип памяти (1 байт), смещение (2 байта): 0...0xFFFF
`sbit`[1]	1	0	0 или 1
`sfr`[1]	8	1	0... 255
`sfr16`[1]	16	2	0... 65535

[1] Типы `bit`, `sbit`, `sfr` и `sfr16` специфичны для компилятора Keil C51. Они не описаны стандартом ANSI, к ним нельзя обращаться через указатели.

[2] Общепринятые сокращения для обозначения некоторых типов данных: `uchar` – `unsigned char`; `uint` – `unsigned int`.

Четырехбайтные числа в формате с плавающей точкой (`float`).

Однобайтовые *типизированные* указатели `data*`, `idata*` для резидентной памяти данных и `pdata*` для одной страницы (256 байтов) внешней памяти данных.

Двухбайтовые *типизированные* указатели `code*` для памяти программ и `xdata*` для внешней памяти данных. Другое название типизированных указателей — *память-зависимые* указатели (англ. *memory-specific pointer*).

Трехбайтовые *нетипизированные* указатели (англ. *generic pointer*), соответствующие стандарту `ANSI C`.

Однобитовые специальные данные (`sbit`) принимают значения 0 или 1, специфичны для `C51`, недоступны через указатели, служат для доступа к отдельным битам регистров специальных функций (англ. *special function register*, SFR).

Однобайтовые специальные данные (`sfr`) служат для доступа к однобайтовым регистрам специальных функций, специфичны для `C51`, недоступны через указатели.

Двухбайтовые специальные данные (`sfr16`) служат для доступа к двухбайтовым регистрам специальных функций, специфичны для `C51`, недоступны через указатели.

1.3. Специальные ключевые слова

В табл. 1.3 ранее был приведен перечень наиболее часто используемых специальных ключевых слов Keil C51, которые позволяют адекватно описать особенности платформы `x51` при программировании. Ниже проведено детальное обсуждение смысла этих ключевых слов (в алфавитном порядке):

`_at_` указывает на необходимость размещения переменной по абсолютному адресу в памяти. Синтаксис следующий:

`тип имя_переменной константа`,

где `тип` определяет тип переменной, `имя_переменной` указывает ее имя, а `константа` указывает адрес размещения переменной. Приведем несколько примеров (символ ';' является частью кода).

```
// Массив символов по адресу xdata 0xB000.
char xdata text[256] _at_ 0xB000;

// Целочисленная переменная i1 по  адресу  xdata 0x9000.
int xdata i1 _at_ 0x9000;

// Переменная порта ввода/вывода по адресу xdata 0xFFE7.
volatile char xdata IO _at_ 0xFFE7;
```

В приведенных примерах переменная `text` определяет массив из 256 символов, который размещается во внешней памяти системы начиная с адреса `0xB000`; переменная `i1` размещается по адресу внешней памяти `0x9000`; переменная `IO`, используемая для хранения данных внешнего устройства ввода/вывода, размещается по адресу `0xFFE7`. В третьем примере переменная объявлена с атрибутом `volatile`, который необходим при работе с внешними портами ввода/вывода;

bit определяет битовый тип переменной, смысл которой очевиден из названия. Битовая переменная может принимать одно из двух значений: 0 или 1. Примеры объявления битовых переменных:

```
bit myBit;
bit testBit (bit flag1, bit flag2);
```

Здесь определены простая битовая переменная `myBit` и функция `testBit`, принимающая в качестве аргументов битовые переменные `flag1`, `flag2` и возвращающая битовое значение. Для хранения битовых переменных используется внутренняя область памяти микроконтроллера `x51`. Здесь имеются определенные ограничения: поскольку эта область памяти имеет размер в 16 байт, то максимальное количество битовых переменных, которые можно разместить в этой области, не превышает 128;

bdata – модификатор памяти, который используется только для объявления переменных. Его нельзя применять для объявления функций. Этот модификатор адресует внутреннюю бит-адресуемую область памяти микроконтроллера `x51`, к которой можно обращаться как к битам (128 битов), так и байтам (16 байтов). Переменные, объявленные с этим ключевым словом, могут быть прочитаны или записаны с использованием битовых инструкций микроконтроллера `x51`. Пример объявления переменной типа `bdata`:

```
uchar bdata bdata_var;
```

code – это модификатор памяти, который используется при объявлении функций и констант, размещаемых в резидентной или внешней памяти программ. Для доступа к функциям, объявленным с этим атрибутом, применяется 16-битовая адресация с выполнением кода `movc a+dptr`, при этом следует учитывать ограничения для памяти программ (64 Кб). Поскольку функции программы в подавляющем большинстве случаев размещаются в памяти программ, то очень редко возникает необходимость использовать этот атрибут. Что же касается констант, то, как правило, с этим атрибутом объявляются табличные данные. Пример объявления константы:

```
uchar code code_constant;
```

compact – директива, устанавливающая *компактную* модель памяти для использования по умолчанию. В этой модели все переменные размещаются во внешней памяти данных. Достигается тот же эффект, как при явном использовании модификатора `pdata`. В этой памяти может быть размещено до 256 байтов. Ограничения появляются вследствие использования косвенной адресации, когда обращение происходит через регистры `r0` и `r1`. Рассматриваемая модель памяти не так эффективна, как малая, и обращение к переменным происходит медленнее. Однако компактная модель все же быстрее, чем большая. Старший бит адреса обычно устанавливается через порт `P2`, причем делается это вручную программистом;

data является модификатором памяти, который используется для доступа к резидентной памяти данных с прямой адресацией (128 байт). Этот класс памяти характеризуется самой быстрой работой с переменными;

far применяется для объявления переменных и констант, для доступа к которым следует использовать 24-битовую адресацию. Максимальное адресное пространство в этом случае ограничивается 16 Мб. Вот примеры использования переменных и констант, объявленных с этим атрибутом:

```
uchar far far_variable;
uchar const far far_const_variable;
```

Фактический диапазон адресов, доступный для адресации с использованием такого типа модификаторов, в значительной степени зависит от архитектуры используемого кристалла. Перед использованием ключевого слова `far` в программах следует внимательно ознакомиться с аппаратными ресурсами конкретного типа используемого микроконтроллера;

idata – это модификатор памяти, который определяет переменные, размещенные во внутренней области памяти микроконтроллера `8051`, доступ к которым осуществляется с использованием 8-битовой косвенной адресации. Следует учитывать, что размер резидентной памяти данных кристалла `8051/52` не превышает 256 байт, при этом нижние адреса памяти типа `idata` могут перекрывать соответствующие адреса памяти, объявленной как `data`. Вот пример объявления переменной типа `idata`:

```
uchar idata variable;
```

interrupt используется для объявления функции, с которой этот атрибут применяется как объявление обработчика прерывания. Детальное обсуждение этого вопроса приведено в п. 2.1.5;

large – директива, устанавливающая *большую* модель памяти для использования по умолчанию. В этой модели все переменные размещаются во внешней памяти данных. Можно также явно указать модель памяти с помощью модификатора `xdata`. Для адресации используется указатель `dptr`. Обращение к памяти через этот указатель не является эффективным, особенно если переменная имеет длину 2 или более байт. При использовании такой модели памяти код получается длиннее, чем при малой или компактной модели;

pdata – это модификатор памяти, который используется для указания переменной, размещенной во внешней памяти данных микроконтроллера `8051` со страничной организацией (размер страницы не превышает 256 байт). Для доступа к переменным в такой области памяти используется косвенная 8-битовая адресация с выполнением кода `movx @Rn`. Пример объявления переменной с атрибутом `pdata`:

```
uchar pdata variable;
```

sbit используется очень часто в программах `x51` для доступа к отдельным битам регистров специальных функций (РСФ). Этот атрибут применяется и в файлах заголовков Keil C51. Пример использования ключевого слова `sbit`:

```
sbit EA = 0xAF;
```

Этот вариант может использоваться и с другой формой записи. Например, для доступа к биту 0 порта `P1` можно создать переменную `bit0` следующим образом:

```
sbit bit0 = P1^0;
```

Подобный вариант объявления битовой переменной, ссылающейся на определенный бит специального регистра, будет часто использоваться в примерах программного кода. Отметим, что переменные такого типа нельзя объявлять в теле функции, они должны быть объявлены вне функции, в которой используются.

sfr и `sfr16` используются для обращения к регистрам специальных функций контроллера `8051`. Например, строка

```
sfr P0 = 0x80;
```

объявляет переменную `P0` и назначает ей адрес специального регистра `0x80`. Это адрес порта `P0`;

small – директива, устанавливающая *малую* модель памяти для использования по умолчанию. В этой модели все переменные размещаются в резидентной памяти данных. Достигается тот же эффект,

как при явном использовании модификатора data. При такой модели обращение к переменным оказывается очень эффективным. Однако все объекты данных и стек должны размещаться в резидентной памяти. Размер стека имеет решающее значение, так как пространство, занимаемое стеком, зависит от глубины вложенности различных функций. Обычно, если компоновщик BL51 сконфигурирован для оверлейной загрузки переменных во внутреннюю память данных, лучше всего использовать малую модель памяти (small);

xdata – модификатор памяти, который используется только для объявления переменных, размещенных во внешней памяти системы 8051 с применением 16-битовой адресации с выполнением ассемблерного кода movx @dptr. Внешняя память, доступная при такой адресации, не превышает 64 Кб. Вот пример объявления переменной типа xdata:

```
uchar xdata variable;
```

Ключевые слова code, data, idata, bdata, xdata, pdata называются *модификаторами памяти*, которые позволяют для каждой переменной точно указать область ее размещения в памяти. Компилятор Keil C51 обеспечивает доступ ко всем областям памяти контроллера. Обращение к резидентной памяти происходит гораздо быстрее, чем к внешней, поэтому переменные, которые используются чаще других, следует размещать в резидентной памяти, а остальные – во внешней. Применяя модификаторы памяти при объявлении переменной, можно указать, где именно она будет размещена. Можно включить сведения о модели памяти в объявление переменной (точно так же, как атрибуты signed и unsigned), например:

```
char data var1;
char code text[] = "Enter parameters:";
unsigned long xdata array[100];
float idata x, y, z;
uint pdata dimension;
uchar xdata vector[10][4][4];
char bdata flags;
```

Если в объявлении переменной модификатор памяти не указан, выбирается модель памяти, установленная по умолчанию. Аргументы функции и переменные класса памяти auto, которые не могут быть размещены в регистрах, тоже хранятся в области памяти, установленной по умолчанию.

Модель памяти, выбираемая в качестве модели по умолчанию, устанавливается директивами компилятора small, compact и large. Почти всегда следует использовать модель памяти small: в этом случае получается самый быстрый, компактный и наиболее эффективный

код. Модели, использующие внешнюю память, стоит использовать только в том случае, если приложение никаким образом не может работать при модели памяти `small`.

Более полная и подробная информация об использовании ключевых слов содержится в документации фирмы Keil на компилятор C51. Кроме того, описание базовых возможностей C51 приводится во встроенной справочной документации компилятора Keil C51.

1.4. Операторы и выражения

Выражение языка программирования Keil C51 – это комбинация знаков операций и операндов, результатом которой является определенное значение, при этом знаки операций определяют действия, которые следует выполнить над операндами. Каждый операнд в выражении также может быть выражением. При вычислении значения выражения учитываются приоритеты операций и изменение порядка выполнения операций при наличии разделителей в виде круглых скобок. Все выражения в Keil C51 формируются и вычисляются по тем же правилам, что и в классическом ANSI C. Необходимо помнить, что неправильное выполнение преобразования типов при вычислении выражений во многих случаях является источником ошибок в программах. При выполнении операций над разными типами данных следует тщательно отслеживать ситуации возможного переполнения или потери значения. В языке Keil C51 предусмотрены следующие арифметические операции:

- суммирование (+);
- умножение (*);
- вычитание (–);
- деление (/);
- вычисление остатка от целочисленного деления (%).

В Keil C51 также предусмотрены *унарные* арифметические операции, которые могут выполняться над одним операндом:

- изменение знака операнда на противоположное (–);
- увеличение (инкремент) значения операнда на единицу (++);
- уменьшение (декремент) значения операнда на единицу (––).

Предусмотрены также побитовые и логические операции:

- побитовое **И** (&);
- логическое **И** (&&);
- побитовое **ИЛИ** (|);
- логическое **ИЛИ** (||);
- инверсия **НЕ** (~);
- отрицание **НЕ** (!=).
- исключающее **ИЛИ** (^);

Побитовые операции широко используются при реализации ввода/вывода через порты данных микроконтроллера `x51`. Логические операции в Keil C51 аналогичны таковым в ANSI C.

Операторы присваивания, условный оператор `if`, `switch`, `case` и операторы цикла `while`, `do` работают в Keil C51 точно так же, как

в стандартном ANSI C. Наибольший интерес для разработчика программного обеспечения x51-совместимых систем представляют расширения языка Keil C51, специально предназначенные для эффективного управления вводом/выводом, обработкой прерываний и манипуляциями с данными в памяти.

1.5. Указатели

Компилятор C51 поддерживает объявление указателей с использованием символа звездочки (*). Указатели можно использовать для выполнения всех операций, доступных в стандартном языке ANSI C. В архитектуре контроллеров x51 различают два вида указателей: *нетипизированные* (англ. *generic pointer*) и типизированные. Другое название типизированных указателей — *память-зависимые* указатели (англ. *memory-specific pointer*).

Нетипизированные указатели имеют размер три байта: в первом байте указывается модель памяти, во втором – старший байт смещения, в третьем – младший байт смещения. Они объявляются так же, как указатели в стандартном C. Приведем примеры нетипизированых указателей:

```
char *s;      /* string ptr */
int  *numptr; /* int ptr    */
long *state;  /* long ptr   */
```

Нетипизированные указатели используются для обращения к любым переменным независимо от их размещения в памяти, что удобно, например, для библиотечных функций. Однако нетипизированные указатели проигрывают типизированным указателям во времени выполнения.

Память-зависимые указатели. При объявлении память-зависимых указателей всегда указывают модификатор памяти. Обращение всегда происходит только к указанной области памяти, например:

```
char data *str;    /* ptr to string in data */
int xdata *numtab; /* ptr to int(s) in xdata */
long code *powtab; /* ptr to long(s) in code */
```

Типизированным указателям не нужен байт, в котором указывается модель памяти, т. к. эта модель определяется во время компиляции, данные указатели могут иметь размер в один байт (указатели idata, data, bdata, и pdata) или в два байта (указатели code и xdata).

Можно существенно ускорить выполнение кода путем использования типизированных указателей. Приведенные ниже примеры программ (табл. 1.5) показывают различия в размерах кода и данных, а также времени выполнения для двух видов указателей.

Таблица 1.5

Сравнение память-зависимых и нетипизированных указателей

Описание	Указатели		
	Память-зависимые		Нетипизированный
	idata	xdata	

Пример программы	`char idata *ip;`
	`char val;`
	`val = *ip;`
		`char xdata *xp;`
		`char val;`
		`val = *xp;`
			`char *p;`
			`char val;`
			`val*p;`

А-код, сгенерированный компилятором Keil C51	`mov r0, ip`
	`mov val, @r0`
		`mov dpl, xp + 1`
		`mov dph, xp`
		`movx a, @dptr`
		`mov val, a`
			`mov r1, p + 2`
			`mov r2, p + 1`
			`mov r3, p`
			`call cldptr`

Размер указателя	1 байт	2 байта	3 байта
Размер кода	4 байта	9 байт	11 байт кода + библ.
Длительность	4 цикла	7 циклов	13 циклов

Преобразование память-зависимых указателей в нетипизированые может проводиться компилятором Keil C51: 1) явным образом путем приведения типов указателей с помощью дополнительного программного кода; 2) неявным образом при передаче параметра функции [4].

Явное преобразование типов указателей требует специального программного кода. Например, пусть `0xA5` – адрес ячейки памяти `idata`, преобразуем его в нетипизированный указатель:

```
uchar *ptr;
ptr = (uchar idata *) 0xA5;
```

При неявном способе компилятор преобразует память-зависимый указатель в нетипизированный, когда память-зависимый указатель передается в качестве аргумента функции, в прототипе которой для этого параметра объявлен нетипизированный указатель:

```
void main (void) {
    WriteMax(0xA5,val);   // 0xA5 - адрес ячейки памяти idata.
}
void WriteMax(uchar xdata *regnum, uchar val) {
    *regnum = val;     // regnum - нетипизированный указатель.
}
```

1.6. Передача параметров и возвращение данных

При передаче параметров из программы в функцию компилятор Keil C51 размещает в регистрах до трех аргументов функций. Это

значительно повышает системную производительность, поскольку аргументы не требуется сохранять в памяти и считывать оттуда. Передачей параметров управляют с помощью специальных директив – `regparms` и `noregparms`. В табл. 1.6 приведен список регистров, используемых для размещения аргументов разных типов.

Таблица 1.6

Регистры, используемые для размещения аргументов разных типов

Количество аргументов	Переменные разных типов			Нетипизированный указатель
	`char`	`int`	`long`, `float`	
1	`r7`	`r6_r7`	`r4-r7`	`r1-r3`
2	`r5`	`r4_r5`	—	—
3	`r3`	`r2_r3`	—	—

Если нет свободных регистров или аргументов слишком много, то для этих аргументов используются фиксированные области памяти.

При возвращении данных регистры микроконтроллера всегда используются для значений, возвращаемых функциями. В табл. 1.7 приведен список типов возвращаемых данных и соответствующих им регистров.

Таблица 1.7

Типы возвращаемых данных и регистры

Тип возвращаемого значения	Регистры	Описание
`bit`	Флаг переноса	
`char, unsigned char`, 1-byte-pointer	`r7`	
`int, unsigned int`, 2-byte-pointer	`r6_r7`	`r6` (H), `r7` (L)
`long, unsigned long`	`r4-r7`	`r4` (H), `r7` (L)
`float`	`r4-r7`	Формат `IEEE` 32 бита
`generic pointer`	`r1-r3`	Тип памяти в `r3`, `r2` (H), r1 (L)

Примечание. H – старший (англ. *high*) и L – младший (англ. *low*) указывают на старшинство байтов в многобайтном слове.

Компилятор Keil C51 выделяет для регистровых переменных до семи регистров процессора (в зависимости от контекста программы). Компилятор имеет информацию о всех регистрах, содержимое которых изменилось в процессе выполнения функции. Компоновщик/загрузчик генерирует один файл на весь проект, содержащий информацию обо всех регистрах, измененных внешними функциями. В результате компилятор знает об изменениях в каждом регистре и может оптимально распределить регистры среди всех функций.

2. Взаимодействие A- и C-программ

Язык программирования Keil C51 поддерживает концепцию структурно-модульного программирования платформы x51. Любая программа на C51 может состоять из одного или нескольких различных модулей. Программный проект может содержать несколько объектных модулей, сохраненных в разных файлах и написанных, возможно, на разных языках программирования, но удовлетворяющих соглашениям о передаче параметров и возвращаемых значениях, принятых в Keil C51 (табл. 1.6, табл. 1.7).

Исходные тексты взаимодействующих объектных модулей могут быть написаны на различных языках программирования, однако основное внимание ниже будет сосредоточено на процессе взаимодействия модулей, написанных на ассемблере Asm51 (A-программы) и C51 (C-программы).

2.1. Соглашения по взаимодействию программ

2.1.1. Настройка стартового адреса C-программы

По-умолчанию компилятор Keil C51 транслирует программу в начальную область резидентной памяти программ со стартовым адресом 0000h. Для перемещения кода программы в другую область памяти программ и изменения стартового адреса необходимо выполнить три действия:

1) изменить стартовый файл (startup code) таким образом, чтобы после аппаратного сброса (reset vector) происходил запуск программы с заданного адреса;

2) разместить векторы прерываний в новой области памяти;

3) настроить редактор связей (linker) на новую область памяти для размещения кода программы.

Приведем конкретные инструкции для перемещения программы во внешнюю область памяти программ со стартовым адресом 2000h. В этих инструкциях все диалоги начинаются с вкладки

Project/Options for Target 'Target1'...,

указание на которую для краткости далее опущено. Более глубокие вкладки показаны в квадратных скобках.

1. Найти стартовый файл ...\c51\lib\starrtup.a51 и скопировать его в свою рабочую директорию под именем sdk-startup.a.

Файл `sdk-startup.a` необходимо отредактировать следующим образом: строку `cseg at 0` заменить на `cseg at 2000h`. Модифицированный файл `sdk-startup.a` необходимо добавить в дерево проекта на первую позицию. Стартовым адресом всегда будет `2000h`.

2. Для настройки векторов прерываний в Keil μVision5 необходимо открыть вкладку [C51], на которой найти элемент

`Interrupt vectors at address:`

и изменить его значение на `0x2000`.

3. Для настройки редактора связей на новую область памяти необходимо открыть вкладку [Target], на которой найти строку

`Off-chip Code memory`

и для ее элемента `Eprom` настроить значения начального адреса новой области памяти `Start=0x2000` и ее размера `Size=0xE000`. После этого открыть вкладку [BL51 Locate] и установить флажок

`Use Memory Layout from Target Dialog`

В завершение необходимо сохранить настройки проекта и повторно выполнить его компиляцию.

2.1.2. Обращение к регистрам в C-программе

Аппаратные ресурсы микроконтроллеров, используемых в разработке, перечисляют в так называемых файлах заголовков (*заголовочных* файлах), имена которых имеют вид `*.h`. Каждому ресурсу в таком файле назначают короткую аббревиатуру, использование которой позволяет сделать код программы наглядным и легко читаемым. Добавление заголовочного файла в исходный текст программы осуществляется оператором

`#include` <*file_name.h*> или `#include` "*file_name.h*",

где *file_name* следует заменить именем конкретного файла заголовков. Имя файла заключают в кавычки: угловые кавычки <...> используются для стандартных файлов заголовков, которые размещены в системной директории `..\c51\inc`. Двойные кавычки "..." используются для нестандартных файлов заголовков пользователя, размещенных в директории проекта или в директории, указанной в маршрутном имени файла.

Некоторые ресурсы являются одинаковыми (стандартными) для разных типов микроконтроллеров одной аппаратной платформы (например, порты `P1`–`P3`, регистры `TCON`, `TMOD` и т. п.). Такие стандартные ресурсы перечисляются в заголовочных файлах базовых микроконтроллеров конкретной аппаратной платформы. Например, заголовочный файл `reg51.h` перечисляет ресурсы базовых микроконтроллеров `80c51` и `80c31` платформы `x51`, а заголовочный файл

reg52.h предназначен для микроконтроллеров 80c52 и 80c32 платформы x52. Для многих задач, выполняемых на базе более продвинутых микроконтроллеров этих платформ, бывает вполне достаточно стандартного заголовочного файла reg51.h (или x52). Сравним, например, фрагменты A- и C-кодов для инициализации последовательного порта микроконтроллера aduc842 (рис. 2.1).

```
#include "aduc842.inc"          #include <aduc842.h>
mov    pllcon, #03h             PLLCON = 0x03;
mov    t3con,  #83h             T3CON  = 0x83;
mov    t3fd,   #2Dh             T3FD   = 0x2D;
mov    scon,   #52h             SCON   = 0x52;
```

Рис. 2.1. Фрагменты A-кода (слева) и соответствующего C-кода (справа) для инициализации последовательного порта микроконтроллера aduc842

Если необходимо использовать дополнительные ресурсы конкретного микроконтроллера, которые отсутствуют в составе базового кристалла, то применяют заголовочные файлы, подготовленные для конкретного микроконтроллера, например, aduc812.h, aduc842.h и т. п. В среде разработки Keil μVision в окне редактора можно нажать правую кнопку мыши и в контекстном меню появится предложение добавить заголовочный файл, наиболее адекватно соответствующий текущему проекту.

2.1.3. Вызов A-подпрограммы из C-программы

В качестве примера рассмотрим для начала простую A-подпрограмму svdisp (рис. 2.2), которая при вызове получает байт данных из аккумулятора, выводит его на линейку светодиодных индикаторов (LED).

```
;------------------------------------------------------------
; Вывод байта данных (A) --> LED
;------------------------------------------------------------
dpp      data   84h            ; Адрес регистра dpp в ADUC.
SV       xdata  07h              ; Адрес LED-регистра ПЛИС.
svdisp:  push   dpp              ; Сохранение регистра dpp.
         push   acc              ; Сохранение регистра acc.
         mov    dpp,#08           ; Выбор 8-й страницы ВПД.
         mov    dptr,#SV              ; Выбор LED-регистра.
         movx   @dptr,a             ; Загрузка LED-регистра.
         pop    acc          ; Восстановление регистра acc.
         pop    dpp          ; Восстановление регистра dpp.
         ret                    ; Возврат из подпрограммы.
;------------------------------------------------------------
```

Рис. 2.2. A-подпрограмма для вывода информации на светодиодные индикаторы

На рис. 2.3 приведен вариант оформления этой A-подпрограммы для вызова из C-программы в качестве C-функции

```
void svdisp(uchar);
```

```
;------------------------------------------------------------
; File: probe2.a
;------------------------------------------------------------
name     myname                      ; 01.
?PR?_svdisp?myname segment code      ; 02.
         public _svdisp              ; 03.
         rseg ?PR?_svdisp?myname     ; 04.
         using   0                   ; 05.
dpp      data   84h                  ; 06.
SV       xdata  07h                  ; 07.
_svdisp: push   dpp                  ; 08.
         push   acc                  ; 09.
         mov    a,r7                 ; 10.
         mov    dpp,#08              ; 11.
         mov    dptr,#SV             ; 12.
         movx   @dptr,a              ; 13.
         pop    acc                  ; 14.
         pop    dpp                  ; 15.
?C0001:  ret                         ; 16.
;------------------------------------------------------------
```

Рис. 2.3. C-функция, полученная из исходной A-подпрограммы

Первые пять строк содержат пять обязательных деклараций:

строка 01. Директива **name** задает имя объектного модуля, генерируемое данной программой. Произвольное имя может иметь длину до сорока символов и записывается в объектный файл вместе с соответствующим модулем. В общем случае имя объектного модуля может не совпадать с именем объектного файла. В исходном файле может быть только одна директива **name**. Если эта директива отсутствует, то в качестве имени объектного модуля используется имя объектного файла без расширения. В примере объектному модулю присвоено имя **myname**;

строка 02. Директива **segment** задает имя перемещаемого сегмента (англ. *generic segment*) и класс памяти для его размещения: в примере класс памяти – **code**, т. е. память программ. Имя перемещаемого сегмента задается в соответствие с соглашением, принятым для компилятора Keil C51, и состоит из трех полей, каждое из которых предваряется символом '**?**'. Первое поле содержит две буквы "**PR**". Второе поле содержит имя A-подпрограммы, которое предваряется символом подчеркивания "**_svdisp**". Третье поле содержит имя объектного модуля "**myname**";

строка 03. Директива **public** перечисляет символьные имена, определенные в модуле, которые редактор связей должен сделать до-

ступными (публичными) для использования в других модулях. В качестве разделителя имен используется запятая. В список не входят имена регистров и сегментов. В примере объявляется публичным имя A-функции `_svdisp`;

`строка 04.` Директива `rseg` объявляет, что перемещаемый сегмент с именем `?PR?_svdisp?myname` отныне является текущим сегментом. Объявление действует до следующего применения директивы `rseg`. Специфицируемый перемещаемый сегмент должен быть ранее создан директивой `segment`;

`строка 05.` Директива `using` назначает текущий банк регистров общего назначения (`r0–r7`). Директива упрощает выбор текущего банка регистров, но результат ее действия может быть переопределен командами переключения банков регистров. Численное значение номера банка регистров лежит в диапазоне от 0 до 3. В соответствии с соглашением, принятым для компилятора Keil C51, передача аргументов (до трех аргументов) между C-программой и C-функцией (A-подпрограммой) может осуществляться в нулевом банке через регистры общего назначения: `r7` (`char`), `r6_r7` (`int`) для первого аргумента `r5` (`char`), `r4_r5` – для второго аргумента и `r3` (`char`), `r2_r3` – для третьего аргумента.

В исходный текст A-подпрограммы внесены изменения:

`строка 08.` Имя A-подпрограммы изменено на `_svdisp`;

`строка 10.` Учтено, что исходный байт из C-программы передается не через аккумулятор, а через регистр `r7`;

`строка 16.` Точка выхода из A-подпрограммы промаркирована меткой `?C0001`.

На рис. 2.4 приведен пример демонстрационной C-программы, которая взаимодействует с A-подпрограммой для визуализации эффекта «бегущие огни». Наблюдаемый эффект – чередование LED-состояний «свечение/затемнение» (*blink*-эффект). Исходное LED-состояние (значение переменной `svet`) может быть изменено в строке 05.

Для компиляции этой C-программы (рис. 2.4) использовался проект Keil μVision, содержащий три файла: `sdk-sturtup.a`, `probe1.c`, `probe2.a`. Приведем краткое описание C-программы:

`строка 01.` Ключевое слово `extern` обозначает, что упомянутая `svdisp` функция является внешней, т. е. она размещена в другом объектном модуле. Остальная часть первой строки является прототипом функции `svdisp`;

`строка 02.` Прототип вспомогательной функции `delay`;

`строка 03.` Начало головной C-программы `main`;

`строка 05.` Объявление байтовой переменной `svet`;

`строка 06.` Организация бесконечного цикла `while`;

```
//-------------------------------------------------------------
// File: probe1.c       Action: blink of LEDs    (Assembler)
//-------------------------------------------------------------
extern void svdisp(unsigned char);          // 01.
void delay(int);                            // 02.
void main(void)                             // 03.
{
    uchar svet = 0;                         // 05.
    while(1){                               // 06.
      svdisp(svet);                         // 07.
      svet = ~svet;                         // 08.
      delay(30000);                         // 09.
    }
}
void delay(int length)                      // 12.
{
    while (length >=0) length--;            // 14.
}
//-------------------------------------------------------------
```

Рис. 2.4. Демонстрационная C-программа, вызывающая A-подпрограмму

`строки 07, 08.` Вывод переменной `svet` в LED-регистр и инверсия значения переменной `svet`;

`строка 09.` Вызов функции задержки с параметром 30 000;

`строка 12.` Подпрограмма для получения задержки;

`строка 14.` Программный цикл, осуществляющий задержку.

2.1.4. Доступ к регистрам ПЛИС из C-программы

Доступ к регистрам ПЛИС стенда SDK1 возможен по прямому адресу для восьмой страницы внешней памяти данных. На рис. 2.5 показан пример демонстрационной C-программы, которая содержит достаточно подробные комментарии. Отметим лишь C-функцию `WriteMax`, которая осуществляет прямую запись в регистр ПЛИС. Эта функция принимает два аргумента: указатель (адрес) регистра ПЛИС на восьмой странице внешней памяти данных (`uchar xdata *regnum`) и байт данных (`uchar val`), который необходимо загрузить в регистр ПЛИС. В подпрограмме-функции приняты меры к сохранению и последующему восстановлению старого номера страницы памяти после переключения на восьмую страницу в регистре `dpp`). Доступ к регистру ПЛИС по указателю `regnum` в C-программе осуществляется с помощью операции звездочка (`*`), т. е. загрузка регистра ПЛИС происходит путем выполнения операции `*regnum = val`. Модификация $\sim$ (инверсия) выводимого байта осуществляется в головной программе, поэтому функция `WriteMax` не возвращает никакого значения. При желании модификацию выводимого байта (инверсию) можно перене-

```
//-------------------------------------------------------------
// File: probe11.c       Action: blink of LEDs       (xdata)
//-------------------------------------------------------------
#include <ADuC842.h>         // Файлы заголовков для aduc842.
#include "sdk_base.h"       // Файлы заголовков для sdk_base.
#define MAXBASE 0x8   // Страница памяти для регистров ПЛИС.
#define SV 0x7        // Адрес LED-регистра на 8-й странице.
void WriteMax(uchar xdata*, uchar); //    Прототип WriteMax.
void delay(int);                    // Прототип функции delay.
void main(void)               // Начало головной С-программы.
{
     uchar svet = 0;          // Переменная для LED-вывода.
     while(1){                             // Бесконечный цикл.
 svet = ~svet;     // Инверсия выводимого байта.
 WriteMax(SV,svet);     // Вывод в регистр ПЛИС.
 delay(30000);  // Задержка с параметром 30 000.
     }
}
//.......Вывод значения val в регистр ПЛИС с адресом regnum.
void WriteMax(uchar xdata *regnum, uchar val)
{
     uchar oldDPP = DPP;           // Сохранение старого DPP.
     DPP = MAXBASE;          // Переключение на 8-ю страницу.
     *regnum = val;                  // Вывод в регистр ПЛИС.
     DPP = oldDPP;   // Восстановление старого значения DPP.
}
void delay(int length)                    // Функция задержки.
{
    while (length >=0) length--;
}                                    // Конец функции delay.
//-------------------------------------------------------------
```

Рис. 2.5. Демонстрационная С-программа, взаимодействующая с регистрами ПЛИС

сти из головной программы в функцию `WriteMax` и дополнить эту функцию возможностью возвращать модифицированный байт. В демонстрационной С-программе (рис. 2.5) использовалась функция задержки `delay`, которая ранее уже применялась (и подробно обсуждалась) в предыдущей демонстрационной программе (рис. 2.4), поэтому функция `delay` повторно здесь не обсуждается.

2.1.5. Программирование прерываний

Рассмотрим пример формирования задержки `delay=1 c` при помощи таймера в режиме прерываний. Для визуализации будем использовать LED-индикаторы, работающие в режиме *blink*-эффекта с периодом этой задержки. На рис. 2.6 приведено решение этой задачи в виде А-программы. Максимальный временной интервал 16-разрядного таймера составляет около 65 мс, поэтому временной интервал `delay=1 c`

```
;-------------------------------------------------------------
; File: probe3.a    Action: blink of LEDs     (A-interrupt)
;-------------------------------------------------------------
dpp      data   84h
pllcon   data  0D7h
SV       xdata  07h
svet     equ    20h                      ; Переменная svet.
icnt     equ    21h;                   ; Счетчик прерываний.
TIME equ (NOT 50000) + 1       ; Константа 3CB0h для 50 мс.
     org  200Bh               ; Вектор прерываний для T/C0.
     ljmp mvector      ; Назначение обработчика прерывания.
     org  2050h               ; Начало головной программы.
     mov pllcon, #03h      ; Тактовая частота 2.097152 МГц.
; --- Инициализация и загрузка C/T0 ---
     mov TCON, #0h                 ; Сброс и останов C/T0.
     mov TMOD, #01h             ; Выбор режима работы C/T0.
     mov TH0, #(HIGH TIME)                 ; Константа 3Ch.
     mov TL0, #(LOW TIME)                  ; Константа B0h.
; --- Инициализация LED-индикаторов ---
     mov svet,#0h
     mov a, svet
     call svdisp
; --- Программирование прерывания ---
     setb ie.1             ; Разрешение прерываний от T/C0.
     setb ie.7        ; Снятие общей блокировки прерываний.
     mov  icnt,#0h     ; Инициализация счетчика прерываний.
     setb tcon.4                       ; Пуск таймера C/T0.
     sjmp $                      ; Конец головной программы.
; --- Обработчик прерывания ---
mvector:
     mov TCON, #0h                 ; Сброс и останов C/T0.
     mov TH0, #(HIGH TIME)                 ; Константа 3Ch.
     mov TL0, #(LOW TIME)                  ; Константа B0h.
     inc icnt              ; Инкремент счетчика прерываний.
     mov a,icnt
     cjne a,#20d,go_ret   ; Переход, если icnt не равен 20.
     mov icnt,#0h             ; Сброс, если icnt равен 20.
     mov a,svet
     cpl a                            ; Инверсия LED-байта.
     mov svet,a
     call svdisp                         ; Вывод LED-байта.
go_ret: setb tcon.4                   ; Пуск таймера C/T0.
     reti                ; Выход из обработчика прерывания.
;-------------------------------------------------------------
```

Рис. 2.6. A-программа обработки прерывания от таймера

формируется с помощью программного цикла из 20 итераций по 50 мс. При этом интервалы по 50 мс формируются таймером `C/T0`. Исходный текст A-подпрограммы **`svdisp`**, используемой для вывода на LED-индикаторы, приведен на рис. 2.2 и обсуждался ранее. Для компиляции этой A-программы (рис. 2.6) использовался проект Keil μVision,

содержащий два файла: `sdk-sturtup.a`, `probe3.a`, исходный текст А-подпрограммы `svdisp` добавляли в файл `probe3.a`.

В компиляторе Keil C51 также предусмотрена возможность обращения к функции обслуживания прерывания (обработчик прерывания) в случае возникновения прерывания. Обработчик прерывания в С-программе имеет следующую типовую структуру (рис. 2.7).

```
//---------------------------------------
void timer0(void) interrupt 1 using 2
{
     ...  ...  ...  ...  ...
}
//---------------------------------------
```

Рис. 2.7. Типовая структура обработчика прерывания в С-программе

В этом примере обработчик прерывания имеет вид С-функции `void timer0(void)`, где `timer0` – имя функции (может быть выбрано любое). В отличие от обычных С-функций функция обработчика прерывания снабжена двумя атрибутами, численные значения которых, записанные правее каждого атрибута, соответствуют номеру прерывания `interrupt` и номеру банка регистров `using`. Второй атрибут (`using`) является необязательным (опциональным) и может быть опущен.

Атрибут `interrupt` – это директива компилятору сформировать программный код, который выполнит следующие действия:

- если в обработчике используются какие-либо регистры из `acc`, `b`, `dph`, `dpl` или `psw`, то их содержимое сохраняется в стеке;
- если не указано ключевое слово `using`, определяющее банк регистров, используемый в обработчике, то рабочие регистры основной программы также сохраняются в стеке;
- перед выходом из обработчика прерывания содержимое предварительно сохраненных в стеке регистров восстанавливается к состоянию перед вызовом прерывания;
- программа-обработчик заканчивается командой `reti`.

При программировании обработчика прерывания необходимо учитывать три существенных ограничения (нарушение любого из них приведет к сообщению об ошибке):

- входные параметры программы-обработчика всегда `void`, т. к. никакие параметры в обработчик прерывания не передаются;
- выходные параметры программы-обработчика всегда `void`, т. к. обработчик прерывания не возвращает никаких параметров;
- функцию-обработчик прерывания нельзя напрямую вызывать из программы, т. к. это, скорее всего, приведет к «краху» (англ. *crash*) программы из-за разрушения программных регистров и стека.

Компилятор Keil C51 поддерживает 32 прерывания с номерами от 0 до 31. Первые 12 номеров прерываний (*N*) для ключевого слова `interrupt` приведены в табл. 2.1. Адреса векторов прерываний (*Вектор*) сопоставлены с встроенным оборудованием кристаллов `i8051`, `aduc812` и `aduc842` [6–8]. Указаны приоритеты обслуживания векторов, назначенные по-умолчанию: чем меньше цифра приоритета, тем выше приоритет.

Таблица 2.1

Номера прерываний для атрибута `interrupt`

Прерывание				Приоритет		
Описание	Источник	*N*	Вектор	`i8051`	`aduc812`	`aduc842`
Внешнее `INT0`	`IE0`	0	`0x003`	1	2	2
Таймер `C/T0`	`TF0`	1	`0x00B`	2	4	4
Внешнее `INT1`	`IE1`	2	`0x013`	3	5	5
Таймер `C/T1`	`TF1`	3	`0x01B`	4	6	6
УАПП	`RI+TI`	4	`0x023`	5	8	8
Таймер `C/T2`	`TF2+EXF2`	5	`0x02B`	—	9	9
АЦП	`ADCI`	6	`0x033`	—	3	3
`I2C+SPI`	`I2CI+ISPI`	7	`0x03B`	—	7	7
Монитор питания	`PSMI`	8	`0x043`	—	1	1
(*Reserved*)	(*Reserved*)	9	`0x04B`	—	—	—
Счетчик `TIC`	`TII`	10	`0x053`	—	—	11
Сторожевой таймер	`WDS`	11	`0x05B`	—	—	2

Из табл. 2.1 видно, что векторы прерывания основного оборудования микроконтроллеров платформы `x51` расположены в диапазоне `0003...005Bh` и попадают в область младших адресов резидентной памяти программ (РПП). В микроконтролерах `aduc812` и `aduc842` пространство РПД соответствует 8 Кб (`0000h...1FFFh`) *Flash*-памяти программ.

В стенде `SDK-1.1` (`aduc812`) пользователь не имеет возможности записи в резидентную *Flash*-память программ. Загрузка программ пользователя осуществляется во внешнюю память программ (ВПП), следовательно, пользователь не может подставить свои процедуры обработки прерываний (точнее, команды перехода к процедурам) по адресам, соответствующим векторам прерываний. Проблема решается следующим образом. В стенде `SDK-1.1` по адресам векторов прерываний в резидентной *Flash*-памяти программ (`0003h...0043h`) записаны команды длинных переходов со смещением `+2000h` на векторы пользовательской таблицы, размещенной в ВПП (`2003h...2043h`).

В стенде `SDK-1.1/s` (`aduc842`) пользователь имеет возможность записи в резидентную *Flash*-память программ. Однако для совместимости программ там также реализован механизм смещения век-

торов в такую же пользовательскую таблицу, размещенную в ВПП (2003h...2043h). Например, вектору прерываний 03h будет соответствовать адрес пользовательской таблицы 2003h.

В рамках данного подхода для назначения обработчику прерывания конкретного вектора прерывания по адресам векторов пользовательской таблицы 20xxh (xxh – вектор прерывания) пользователем размещаются команды длинных переходов на обработчик прерываний.

Атрибут interrupt 1 (см. рис. 2.7) предписывает компилятору Keil C51 в числе прочих действий сформировать также фрагмент кода, обеспечивающий длинный переход от вектора прерывания пользовательской таблицы к обработчику прерывания. В обсуждаемом примере (см. рис. 2.7) дополнительный код в ассемблерном виде будет выглядеть примерно следующим образом (рис. 2.8, *а*).

```
                    200B 02         200B 02
org 200Bh           200C 20         200C 0E
Ljmp timer0         200D 0E         200D 20
```

а *б* *в*

Рис. 2.8. Дополнительный код, вызываемый атрибутом interrupt 1: *а* – ассемблерный код; *б* – бинарный код компилятора Keil C51; *в* – бинарный код компилятора SDCC

Однако бинарный код может зависеть от конкретной реализации компилятора Keil C51. Во всех случаях по адресу 200Bh будет записан двоичный код 02h, соответствующий коду команды Ljmp. Следом за кодом команды Ljmp должен идти двухбайтовый адрес перехода в формате *big-endian*: сначала старший байт адреса перехода, а затем младший байт адреса перехода (пусть для примера адрес перехода будет 200Eh). Компилятор Keil C51 работает в коде, указанном на рис. 2.8, *б*. Альтернативный компилятор SDCC работает в коде *little-endian* (рис. 2.8, *в*), поэтому для компилятора SDCC требуется дополнительный программный код, исправляющий запись адреса перехода к обработчику прерываний. Обычно этот программный код оформляют в виде С-функции. На рис. 2.9 приведен код такой С-функции с именем SetVector. Ее вызов происходит следующим образом:

```
SetVector(0x200B, (void *) timer0);
```

Следует отметить, что при работе в среде Keil μVision нет необходимости использовать функцию SetVector.

На рис. 2.10 приведена С-программа обработки прерывания от таймера C/T0, выполняющая те же самые действия, что и ранее обсуждаемая А-программа (рис. 2.6). С-программа (рис. 2.10) рассчитана на использование компилятора Keil μVision и не использует функцию SetVector. Номера строк не являются частью программного кода, строки программы пронумерованы только для удобства обсуждения:

```
//-------------------------------------------------------------
#include "sdk_base.h"
void SetVector(uchar xdata* Address, void* Vector)
{
     uchar xdata * TmpVec;           // Временная переменная.
     *Address = 0x02;          // 02h - это код команды Ljmp.
     TmpVec = (uchar xdata *) (Address + 1);
                              // Старший байт адреса перехода.
     *TmpVec = (uchar) ((unsigned short)Vector >> 8);
     ++TmpVec;
     *TmpVec = (uchar) Vector;               // Младший байт.
}
//-------------------------------------------------------------
```

Рис. 2.9. Типовая структура C-функции SetVector

строки 00, 01 – обращение к заголовочным файлам **aduc842.h** и **sdk_base.h**;

строки 02...04 – расчет констант **LowTime** и **HighTime** для загрузки в двухбайтный регистр данных таймера **C/T0**. Расчет выполняется средствами препроцессора в два этапа: на первом этапе (*строка 01*) получают дополненный до двойки код константы **TIME**, которая соответствует времени задержки в микросекундах; на втором этапе из этого двухбайтного кода выделяют младший (*строка 02*) и старший (*строка 03*) байты для загрузки в таймер **C/T0**;

строки 05, 06 – прототипы внешней функции **svdisp** и обработчика прерывания **timer0**;

строки 07, 08 – декларация глобальных переменных **svet** для вывода на LED-индикаторы и **icnt** для подсчета количества выполненных прерываний;

строки 09, 10 – начало головной программы **main**;

строка 11 – настройка тактовой частоты, чтобы одно состояние таймера соответствовало 1 мкс;

строки 12...15 – инициализация и загрузка таймера **C/T0**;

строки 16...17 – инициализация переменных **svet** и **icnt**;

строка 18 – вывод переменной **svet** на LED-индикаторы;

строки 19, 20 – разрешение прерываний;

строка 21 – пуск таймера **C/T0**;

строка 22 – завершение головной программы бесконечным циклом **while(1)** для предотвращения выхода из программы;

строки 24, 25 – начало обработчика прерывания **timer0**.

```
//-------------------------------------------------------------
#include <aduc842.h>                                        // 00.
#include "sdk_base.h"                                       // 01.
#define TIME 50000              // (NOT 50000) + 1 = 0x3CB0. 02.
#define LowTime  ((~TIME + 1) & 0x0FF)    // Константа 0xB0. 03.
#define HighTime ((~TIME + 1)>>8)         // Константа 0x3C. 04.
extern uchar svdisp(uchar);                                 // 05.
void timer0(void);          // Прототип функции обработчика. 06.
uchar svet;            // Переменная svet для вывода на LED. 07.
uint icnt;                 // Счетчик количества прерываний. 08.
void main (void)               // Начало головной программы. 09.
{                                                           //  10.
     PLLCON = 3;           // Тактовая частота 2.097152 МГц. 11.
     TCON   = 0;            // Останов и сброс таймера C/T0. 12.
     TH0    = HighTime;     // Загрузка старшего байта C/T0. 13.
     TL0    = LowTime;      // Загрузка младшего байта C/T0. 14.
     TMOD   = 1;              // Задание режима работы C/T0. 15.
     svet   = 0;           // Инициализация переменной svet. 16.
     icnt   = 0;       // Инициализация счетчика прерываний. 17.
     svdisp(svet);          // Вывод переменной svet на LED. 18.
     ET0    = 1;            // Разрешение прерывания от TF0. 19.
     EA     = 1;      // Снятие общей блокировки прерываний. 20.
     TR0    = 1;                        // Пуск таймера C/T0. 21.
     while(1);                  // Конец головной программы. 22.
}                                                           //  23.
void timer0(void) interrupt 1      // Обработчик прерывания. 24.
{                                                           //  25.
     TCON   = 0;            // Останов и сброс таймера C/T0. 26.
     TH0    = HighTime;     // Загрузка старшего байта C/T0. 27.
     TL0    = LowTime;      // Загрузка младшего байта C/T0. 28.
     ++icnt;               // Инкремент счетчика прерываний. 29.
     if(icnt == 20){   // Если выполнено 20 прерываний, то:  30.
       icnt = 0;        //      сбросить счетчик прерываний; 31.
       svet = ~svet;    //    инвертировать переменную svet; 32.
       svdisp(svet);    //   вывести переменную svet на LED. 33.
     }                                                      //  34.
     TR0 = 1;                           // Пуск таймера C/T0. 35.
}                           // Конец обработчика прерывания. 36.
//-------------------------------------------------------------
```

Рис. 2.10. C-программа обработки прерывания от таймера

Атрибут `interrupt` с параметром `1` указывает на вектор прерывания `000Bh` (или `200Bh` в пользовательской таблице);

строки 26, 28 – останов и реинициализация таймера `C/T0`;

строки 29, 30 – инкремент счетчика прерываний `icnt` и проверка, что его значение не достигло двадцати. В таком случае произойдет переход на строку `34`;

строки 31...33 – если значение `icnt` = 20, то сбрасывается счетчик прерываний, инвертируется значение переменной `svet` и обновленное значение переменной `svet` выводится на `LED`-индикаторы.

Отметим, что обновленное значение переменной `svet` может быть получено также какой-либо другой арифметической или логической операцией. Операция *инверсия* выбрана только в качестве примера;

`строка 35` – пуск таймера `C/T0` и выход из обработчика прерываний по команде `reti`.

2.2. Доступ к стандартным библиотекам

Компилятор Keil C51 содержит 6 оптимизированных библиотек, которые размещены в 7 библиотечных файлах (табл. 2.2), подключаемых во время компиляции. Имена шести файлов начинаются с символов `C51`. Модели памяти кодируются следующими буквами: **S** – малая (англ. *small*), **C** – компактная (англ. *medium*) и **L** – большая (англ. *large*). Наличие поддержки операций для чисел в формате с плавающей точкой кодируется буквами `FP` (англ. *floating point*, FP). Седьмой файл содержит оптимизированную библиотеку для микроконтроллера `Philips 8xC751` и его производных. Все исходные коды используются выполняющими аппаратно-зависимый ввод-вывод библиотечными модулями и размещаются в каталоге `\c51\lib`. С помощью этих файлов можно использовать библиотеки для выполнения операций ввода-вывода при работе с любыми периферийными устройствами.

Таблица 2.2

Файлы библиотек

Файлы библиотеки	Модель памяти	Поддержка операций для чисел с ПТ
`C51S.LIB`	Small	–
`C51FPS.LIB`	Small	+
`C51C.LIB`	Medium	–
`C51FPC.LIB`	Medium	+
`C51L.LIB`	Large	–
`C51FPL.LIB`	Large	+
`80C751.LIB`	Библиотека для контроллера Philips 8xC751 и его производных	

Библиотечные файлы (в совокупности со встроенными функциями) предоставляют пользователю свыше 100 различных библиотечных функций (табл. 2.3), помимо стандартного набора из ANSI C. В руководстве по компилятору Keil C51 [4] содержится детальное описание библиотечных функций, правила их параметризации при вызове и примеры программного кода с их использованием.

Некоторые функции, приведенные в табл. 2.3, являются *встроенными* (*подставляемыми*) функциями, которые представляют собой

Таблица 2.3

Библиотечные функции Keil C51

Функция	Функция	Функция	Функция
chkfloat	cosh	memccpy	strrchr
crol	exp	memchr	strrpbrk
cror	fabs	memcmp	strrpos
_getkey	floor	memcpy	strspn
irol	free	memmove	tan
iror	getchar	memset	tanh
lrol	gets	modf	toascii
lror	init_mempool	pow	toint
nop	isalnum	printf	strcpy
testbit	isalpha	putchar	strcspn
_tolower	iscntrl	puts	strlen
_toupper	isdigit	rand	strncat
abs	isgraph	realloc	strncmp
acos	islower	scanf	strncpy
asin	isprint	setjmp	strpbrk
atan	ispunct	sin	strpos
atan2	isspace	sinh	tolower
atof	isupper	sprintf	toupper
atoi	isxdigit	sqrt	ungetchar
atol	labs	srand	va_arg
cabs	log	sscanf	va_end
calloc	log10	strcat	va_start
ceil	longjmp	strchr	vprintf
cos	malloc	strcmp	vsprintf

макроопределения. Макроопределения хранятся не в библиотечных файлах, а встроены в компилятор. При компиляции подставляемых функций генерируется встроенный (*in-line*) код, который при выполнении программного кода будет более эффективен, чем таковой при вызове библиотечных процедур путем использования инструкций `ACALL` или `LCALL`. Встроенные функции компилятора Keil C51 перечислены в табл. 2.4.

Прототипы библиотечных функций сгруппированы по нескольким заголовочным файлам (англ. *include files*). Приведем краткое описание лишь некоторых заголовочных файлов:

absacc.h содержит макроопределения (макросы), предоставляющие доступ к различным областям памяти (12 макроопределений). Названия макросов принято записывать прописными буквами:

```
CBYTE       CWORD       DBYTE       DWORD
FARRAY      FCARRAY     FCVAR       FVAR
PBYTE       PWORD       XBYTE       XWORD
```

ctype.h содержит заголовки функций для процедур, распознающих и преобразующие ASCII-символы (17 функций):

```
isalnum   isalpha   iscntrl   isdigit   isgraph
islower   isprint   ispunct   isspace   isupper
isxdigit  toascii   toint     tolower   _tolower
toupper   _toupper
```

intrins.h содержит заголовки для встроенных (представляемых) функций (табл. 2.4) и функции _chkfloat_ (9 функций):

Таблица 2.4

Встроенные функции компилятора Keil C51

Встроенные функции	Описание
crol	Циклический сдвиг символа влево
cror	Циклический сдвиг символа вправо
irol	Циклический сдвиг целого числа влево
iror	Циклический сдвиг целого числа вправо
lrol	Циклический сдвиг длинного целого числа влево
lror	Циклический сдвиг длинного целого числа вправо
nop	Нет операции (инструкция 8051 NOP)
testbit	Проверить и очистить бит (инструкция 8051 JBS)

math.h содержит заголовки для всех процедур, осуществляющих математические вычисления с числами с плавающей точкой, а также для некоторых других математических процедур (25 функций):

```
abs       acos      asin      atan      atan2
cabs      ceil      cos       cosh      exp
fabs      floor     fmod      fprestore fpsave
labs      log       log10     modf      pow
sin       sinh      sqrt      tan       tanh
```

stdio.h содержит заголовки и определения процедур ввода-вывода (12 функций), а также определение константы EOF:

```
getchar      _getkey      gets         printf
putchar      puts         scanf        sprintf
sscanf       ungetchar    vprintf      vsprintf
```

stdlib.h содержит заголовки и определения для процедур преобразования типов и выделения памяти (13 функций), а также определение константы NULL:

```
atof         atoi      atol      calloc    free
init_mempool malloc    rand      realloc   srand
strtod       strtol    strtoul
```

string.h содержит заголовки и определения для процедур манипулирования строками и буферами (21 функция), а также определение константы NULL:

```
memccpy      memchr    memcmp    memcpy    memmove
memset       strcat    strchr    strcmp    strcpy
strcspn      strlen    strncat   strncmp   strncpy
strpbrk      strpos    strrchr   strrpbrk  strrpos
strspn
```

2.3. Дополнительные возможности

2.3.1. Поддержка языка PL/M-51

Популярный язык программирования PL/M-51 был разработан фирмой Intel и во многом схож с языком Keil C51, поэтому программы, написанные на C и PL/M-51, легко взаимодействуют друг с другом. В C-программе можно легко объявить функции PL/M-51, добавив модификатор alien. Кроме того, в C-программе доступны все глобальные переменные, объявленные в модуле на PL/M-51, например:

```
extern alien char plm_func (int, char);
```

Инструментальная среда Keil μVision и компилятор PL/M-51 генерируют объектные файлы в одном и том же формате OMF51, поэтому все символические ссылки оказываются разрешенными.

2.3.2. Реентерабельные функции

Реентерабельные функции – это такие функции, которые могут вызываться одновременно разными процессами. Особенность в том, что при выполнении такой функции другой процесс может ее вызвать повторно и начать ее выполнение. Обычные функции Keil C51 не могут быть вызваны рекурсивно или повторно до окончания выполнения, поскольку аргументы функций и локальные переменные хранятся в фиксированных областях памяти. Использование атрибута `reentrant` дает возможность объявить функцию как реентерабельную, которая далее может вызываться рекурсивно, например:

```
int calc (char i, int b) reentrant
{
   int x;
   x = table [i];
   return (x*b);
}
```

Реентерабельные функции разрешено вызывать рекурсивно или также одновременно двумя разными процессами. Данные функции можно использовать в приложениях, осуществляющих управление в режиме реального времени, а также когда функцию необходимо вызывать как программно, так и по прерыванию. Стек реентерабельной функции может размещаться в резидентной или внешней памяти в зависимости от модели памяти.

Атрибут `reentrant` дает возможность объявить реентерабельной лишь конкретную функцию, т. е. локальный участок программы, поскольку объявление всей программы реентерабельной может потребовать неприемлемо больших ресурсов.

3. Библиотека функций sdk_base

Описана библиотека sdk_base, скомпонованная на основе ассемблерных модулей, которые обеспечивают пользователю доступ к различным периферийным устройствам стендов SDK-1.1 и SDK-1.1/s. Проектирование, работа и примеры использования этих ассемблерных модулей подробно обсуждались ранее в учебном пособии по микропроцессорной технике [2]. В настоящем разделе показана возможность доработки этих модулей до A-функций, вызываемых из C-программы, и компоновка их в единый библиотечный модуль sdk_base (24 функции). В рассматриваемом случае (прил. Б) речь идет не о стандартных библиотеках, которые обсуждались в предыдущем разделе, а о библиотеке и библиотечных функциях, созданных пользователем.

3.1. Библиотечные A-функции доступа к оборудованию

Для получения доступа к библиотечным функциям файл библиотеки следует сначала подключить к своему проекту. В окне проекта (Project Window) IDE Keil μVision 5 должна быть получена примерно следующая итоговая картина, отображающая дерево проекта:

```
[-] Project: probe
   [-] Target 1
      [-] Source Group 1
             sdk-sturtup.a
         [+] sdk-main.c
             sdk_base.lib
```

В этом примере probe – это имя проекта; sdk-main.c – головная C-программа; sdk-sturtup.a – стартовый файл; sdk_base.lib – файл библиотеки, содержащий библиотечные функции.

Заголовочный файл sdk_base.h, содержащий прототипы библиотечных функций и определения некоторых важных констант, должен быть объявлен (подключен) в головной C-программе:

```
#include "sdk_base.h"
void main (void)
{
  ...  ...  ...
}
```

После этого библиотечные функции и предопределенные в заголовочном файле константы (табл. Б.1 и Б.2, прил. Б) становятся доступными для использования в C-программе.

3.1.1. Функции досупа к регистрам ПЛИС

Рассмотрим функции библиотеки для доступа к регистрам ПЛИС:

putbyte служит для вывода одного байта в указанный регистр ПЛИС. Формат (прототип) вызова С-функции имеет следующий вид:

```
void putbyte (uchar, uchar);
```

Первый аргумент – выводимый байт, второй аргумент задает адрес регистра ПЛИС. В заголовочном файле определены константы, соответствующие адресам регистров ПЛИС:

- KB – регистр клавиатуры;
- DATA_IND – шина данных ЖК;
- EXT_LO – внешний порт (Low);
- EXT_HI – внешний порт (High);
- C_IND – регистр команд ЖК;
- SV – светодиодные индикаторы;
- ENA – управление внешними портами, звуком и INT0;

svdisp служит для вывода одного байта на линейку светодиодных индикаторов. Нулевые значения битов соответствуют погашенным (выключенным) светодиодам, а единичные – светящимся (включенным) светодиодам. В С-программе используется следующий формат (прототип) для вызова данной функции:

```
void svdisp (uchar);
```

Исходный ассемблерный текст функции `svdisp` обсуждался ранее и приведен на рис. 2.3, а пример ее вызова из С-программы показан на рис. 2.4;

getbyte служит для ввода одного байта из указанного регистра ПЛИС. Формат (прототип) вызова данной С-функции имеет следующий вид:

```
uchar getbyte (uchar);
```

Аргумент функции задает адрес регистра ПЛИС. Для задания адресов возможно использование констант, определенных выше для функции `putbyte`;

lputchar служит для вывода одного байта на вход жидкокристаллического (ЖК) дисплея (англ. *liquid crystal display*, LCD) стенда. В качестве выводимого байта может быть код символа для отображения на ЖК-дисплее (режим *данных*) или код команды для управления работой ЖК-дисплея (режим *команд*). Прототип такой функции `lputchar` имеет следующий вид:

```
void lputchar (uchar, uchar);
```

Первый аргумент – выводимый байт, второй аргумент задает значение бита RS, определяющего режим вывода: 0 – команды, 1 – данные.

При выводе данных необходимо учитывать, что в диапазоне кодов `0x0...0x7F` (первая страница) знакогенератор ЖК-дисплея полностью совместим со стандартной ASCII-кодировкой. Для второй страницы знакогенератора (`0x80...0xFF`) используется нестандартная

кодировка символов, т. е. при выводе кириллицы и символов псевдографики следует использовать таблицу кодов знакогенератора [5].

В заголовочном файле `sdk_base.h` определены константы:

- `LCD_BF = 0x80` – маска для выделения старшего бита в байте;
- `LCD_CLR` – код команды очистка экрана ЖК-дисплея;
- `LCD_CR` – код команды возврата курсора в начало строки;
- `LCD_POS(N)` – макроопределение для вычисления константы, соответствующей коду команды для перемещения курсора в позицию `N`. Вычисление производится по формуле `LCD_POS(N) = N | 0x80`.

Код курсора соответствуют номеру знакоместа ЖК-дисплея `N`, отсчитанному слева направо:

- `0x00...0x0F` для верхней строки дисплея;
- `0x40...0x4F` для нижней строки дисплея;

lgetchar служит для ввода байта из выходного порта ЖК-дисплея (`RS = 0; RW = 1`). Вводимый байт содержит информацию о текущем положении курсора ($a_0 \ldots a_6$) и состоянии флага занятости `BF` (a_7):

BF	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Значение флага `BF = 1` означает, что ЖК-дисплей занят обработкой предыдущего байта данных или команд и не может принять новый байт. Если `BF = 0`, то ЖК-дисплей готов к приему нового байта данных или команд. Координаты текущего положения курсора представлены в кодировке, принятой для команды `lputchar`. Прототип функции `lgetchar` имеет следующий вид:

```
uchar lgetchar (void);
```

В табл. 3.1 приведены значения констант и обозначения флагов, использованных в функциях `lputchar` и `lgetchar`.

Таблица 3.1

Константы и флаги, использованные в функциях `lputchar` и `lgetchar`

Константа	Память	Значение	Флаг	Формат	Место
MAXBASE	xdata	0x08	E	bit	acc.0
DATA_IND	xdata	0x01	RW	bit	acc.1
C_IND	xdata	0x06	RS	bit	acc.2

Примечание. `MAXBASE` – номер страницы памяти, где расположены регистры ПЛИС; `DATA_IND` (данные) и `C_IND` (команды) – адреса регистров ЖК-дисплея; E (строб), RW (чтение/запись), RS (команды/данные) – флаги управления ЖК-дисплеем.

Компилятор Keil C51 добавляет символ подчеркивания ('_') в начало имени функции, чтобы указать, что все аргументы передаются

через регистры [4], поэтому при оформлении ассемблерных модулей следует соблюдать важное правило: имя A-функции дополняется слева символом подчеркивания '_' в том случае, когда список передаваемых аргументов не пустой (не `void`). Если список передаваемых аргументов пустой (`void`), то имя A-функции не следует дополнять символом подчеркивания '_'. Примером этого являются имена A-функций на рис. 3.1:

`_lputchar` для C-фукции `void lputchar (char, char)`;
`lgetchar` для C-фукции `char lgetchar (void)`.

На рис. 3.1 приведены исходные тексты ассемблерных модулей, соответствующих функциям `lputchar` и `lgetchar`.

```
;----------------------------   ;----------------------------
_lputchar: push  dpp            lgetchar: push  dpp
           push  dph                      push  dph
           push  dpl                      push  dpl
           push  acc                      push  acc
           mov   dpp,#MAXBASE             mov   dpp,#MAXBASE
           mov   a,r7                     mov   a,#00000010b
           mov   dptr,#DATA_IND           mov   dptr,#C_IND
           movx  @dptr,a                  movx  @dptr,a
           mov   a,#00000100b             nop
           cjne  r5,#0h,put_m1            setb  E
           clr   RS                       movx  @dptr,a
   put_m1: mov   dptr,#C_IND              mov   dptr,#DATA_IND
           movx  @dptr,a                  movx  a,@dptr
           setb  E                        mov   r7,a
           movx  @dptr,a                  clr   E
           clr   E                        mov   dptr,#C_IND
           movx  @dptr,a                  movx  @dptr,a
           pop   acc                      pop   acc
           pop   dpl                      pop   dpl
           pop   dph                      pop   dph
           pop   dpp                      pop   dpp
           ret                            ret
;----------------------------   ;----------------------------
```

Рис. 3.1. Исходные коды A-функций `lputchar` и `lgetchar`

getkey служит для опроса клавиатуры стенда, получения кода нажатой клавиши (`0x00` ... `0x0F`) или кода `0xFF`, соответствующего отсутствию нажатых клавиш. Прототип функции `getkey` имеет вид:

```
uchar getkey(void);
```

В табл. 3.2 приведены коды нажатых клавиш, на рис. 3.2 показан исходный текст ассемблерного модуля функции `getkey`.

Функция `getkey` не осуществляет подавление *дребезга* контактов. Один из возможных способов программного подавления этого нежелательного явления — контроль двойного события «нажатие кла-

Таблица 3.2

Коды клавиш, возвращаемые функцией `getkey`

Возвращаемый код	Клавиша	Возвращаемый код	Клавиша
0x00	1	0x08	7
0x01	2	0x09	8
0x02	3	0x0A	9
0x03	A	0x0B	C
0x04	4	0x0C	*
0x05	5	0x0D	0
0x06	6	0x0E	#
0x07	B	0x0F	D

виши – отпускание клавиши». Алгоритм контроля состоит из четырех шагов:

- ожидание нажатия клавиши;
- запоминание кода нажатой клавиши;
- ожидание возврата клавиши в исходное состояние;
- возвращение запомненного кода.

Во многих случаях этого алгоритма вполне достаточно для реше-

```
;----------------------------
getkey:  push  dpp
         push  dph
         push  dpl
         push  acc
         push  b
         push  1h
         mov   dpp,#MAXBASE
         mov   dptr,#KB
         mov   r1,#0h
         mov   b,#01111111b
kb_lp:   mov   a,b
         rl    a
         mov   b,a
         movx  @dptr,a
         movx  a,@dptr
         cpl   a
         anl   a,#0F0h
         jnz   kb_cod
         inc   r1
               ...
                                        ...
                                cjne  r1,#4,kb_lp
                                mov   a,#0FFh
                                sjmp  kb_end
                      kb_cod:   mov   b,#0h
                                mov   c,acc.7
                                orl   c,acc.5
                                mov   b.2,c
                                mov   c,acc.7
                                orl   c,acc.6
                                mov   b.3,c
                                mov   a,b
                                add   a,r1
                      kb_end:   mov   r7,a
                                pop   1h
                                pop   b
                                pop   acc
                                pop   dpl
                                pop   dph
                                pop   dpp
                                ret
                     ;------------------------
```

Рис. 3.2. Исходные коды A-функции `getkey`

ния проблемы программного подавления дребезга контактов клавиатуры. Приведем фрагмент С-кода, реализующего данный алгоритм:

```
// Ожидание нажатия какой-либо клавиши на клавиатуре стенда.
     while((symbol = getkey()) == 0xFF);
// Возвращаемый результат запоминается в переменной symbol.
// Ожидание отпускания ранее нажатой клавиши.
     while(getkey() != 0xFF);
```

3.1.2. Внешний параллельный порт ПЛИС

put_lpt служит для вывода двухбайтового слова во внешний 16-разрядный порт ПЛИС. Прототип функции put_lpt имеет вид:

```
void put_lpt(uint);
```

get_lpt служит для ввода двухбайтового слова из внешнего 16-разрядного порта ПЛИС. Прототип функции get_lpt имеет вид:

```
uint get_lpt(void);
```

В программе на С51 вводимые и выводимые двухбайтовые данные внешнего порта описываются форматом uint.

3.1.3. Последовательный порт UART

uart_ini проводит начальную инициализацию последовательного порта контроллера aduc812 (или aduc842) для работы на скорости 9600 бит/с. Прототип функции uart_ini имеет вид:

```
void uart_ini(uchar);
```

Аргумент данной функции может принимать одно из двух значений: 0 (aduc812) или 1 (aduc842). В заголовочном файле определены две соответствующие константы: ADUC812 и ADUC842. На рис. 3.3 показан исходный текст ассемблерного модуля функции uart_ini;

get_uart ожидает приема байта контроллером UART и доставляет программе этот байт в качестве кода возврата. Ожидание прихода байта осуществляется путем программного опроса флага приемника UART. Прототип функции get_uart имеет вид:

```
uchar get_uart (void);
```

put_uart осуществляет передачу байта контроллером UART. Ожидание конца передачи осуществляется путем программного опроса флага передатчика UART. Прототип функции put_uart имеет вид:

```
void put_uart (uchar);
```

На рис. 3.4 показаны исходные тексты ассемблерных модулей функций get_uart и put_uart;

get_idata вспомогательная функция, служащая для чтения содержимого адресованной ячейки памяти idata. Аргументом функции

```
;-------------------------------------------------------------
_uart_ini:cjne   r7,#0h,uar_m1
          mov    th1,#S9600    ; 0FDh
          orl    tmod,#20h
          anl    pcon,#7Fh
          orl    tcon,#40h
          mov    scon,#50h
          clr    ie.4
          ret
  uar_m1: mov    pllcon,#03h   ; Core 2.097152 MHz
          mov    t3con,#83h
          mov    t3fd,#2Dh
          mov    scon,#52h
          clr    ie.4
          ret
;-------------------------------------------------------------
```

Рис. 3.3. Исходный текст ассемблерного модуля функции `uart_ini`

```
;---------------------          ;---------------------
get_uart:jnb scon.0,$          _put_uart:clr scon.1
         mov r7,sbuf                     mov sbuf,r7
         clr scon.0                      jnb scon.1,$
         ret                             clr scon.1
                                         ret
;---------------------          ;---------------------
```

Рис. 3.4. Исходные тексты ассемблерных модулей функций `get_uart` и `put_uart`

является адрес ячейки памяти `idata`, содержимое ячейки доставляется в виде возвращаемого значения. Прототип функции `get_idata` имеет вид:

```
uchar get_idata (uchar);
```

put_idata вспомогательная функция, которая служит для записи в адресованную ячейку памяти `idata`. Первым аргументом функции является адрес ячейки памяти `idata`, второй аргумент – новое содержимое этой ячейки памяти. Прототип функции `put_idata` имеет вид:

```
void put_idata(uchar, uchar);
```

На рис. 3.5 показаны исходные тексты ассемблерных модулей функций `get_idata` и `put_idata`.

3.1.4. Последовательный порт I2C

В разделе описаны функции библиотеки `sdk_base`, предназначенные для иллюстрации взаимодействия через интерфейс `I2C` с ведомыми устройствами (`slave`) и демонстрирующие примеры работы с интерфейсом `I2C` учебных микроконтроллерных стендов `sdk-1.1`

```
;---------------------          ;---------------------
_get_idata:                     _put_idata:
      push 0h                         push 0h
      mov  a,r7                       mov  a,r7
      mov  r0,a                       mov  r0,a
      mov  a,@r0                      mov  a,r5
      mov  r7,a                       mov  @r0,a
      pop  0h                         pop  0h
      ret                             ret
;---------------------          ;---------------------
```

Рис. 3.5. Исходные тексты ассемблерных модулей функций get_idata и put_idata

(aduc812) и sdk-1.1/s (aduc842). Функции рассчитаны на взаимодействие с ведомыми устройствами, имеющими 8-разрядное внутреннее адресное пространство. Контроллер I2C при этом работает в режиме ведущего устройства (master). Ввиду того, что программный текст ассемблерных модулей достаточно длинный и содержит большое количество вызовов более мелких рутинных процедур, исходные тексты A-функций не приводятся. Рассмотрим библиотечные функции:

getack служит для проверки готовности (пинг) ведомого устройства (slave) к обмену данными; ее прототип имеет вид:

```
bit getack (uchar);
```

Аргумент функции содержит адрес ведомого I2C-устройства, возвращаемое значение – флаг готовности ведомого устройства: 0 – устройство не готово; 1 – устройство готово к обмену данными. Вызов данной библиотечной функции должен предшествовать любому обращению к I2C-устройству;

receiveblock позволяет получить блок данных от ведомого (slave) I^2C-устройства с 8-битным внутренним адресным пространством. Блок данных при получении размещается в резидентной памяти данных (idata). Прототип функции receiveblock имеет вид:

```
bit receiveblock(uint, uchar, uchar);
```

Первый аргумент (uint, два байта) содержит адрес I2C-устройства (старший байт) и адрес начала блока-источника во внутреннем адресном пространстве I2C-устройства (младший байт). Второй аргумент (uchar, один байт) содержит адрес буфера в области idata, где будет размещен принятый блок. Третий аргумент (uchar, один байт) задает длину блока байтов, т. е. количество получаемых байтов.

При успешном завершении процедуры возвращаемое значение (код возврата) равно нулю. При любой ошибке во время процедуры код возврата будет единичным. По значению кода возврата возможна организация обработки ошибки. Если обработка ошибки не планируется, то анализ состояния флага можно исключить;

sendblock позволяет передать блок данных в ведомое (slave) I^2C-устройство с 8-битным внутренним адресным пространством. Блок данных для отправки должен быть предварительно размещен в резидентной памяти данных (idata). Прототип функции sendblock имеет вид:

```
bit sendblock(uint, uchar, uchar);
```

Передаваемые параметры и коды возврата (за исключением второго аргумента) совпадают с таковыми для функции receiveblock. Второй аргумент содержит адрес буфера в области idata, где предварительно должен быть размещен блок данных, подготовленный для пересылки в адресуемое I2C-устройство.

gettime служит для демонстрации функции receiveblock. С помощью gettime получают четыре байта из ведомого I2C-устройства (микросхема часов в составе стенда). Четыре полученные байта размещаются в буфере (область памяти idata) и соответствуют четырем байтам времени в упакованном двоично-десятичном коде (ДДК):

- первый (младший) байт, ms – десятые и сотые доли секунды;
- второй байт, ss – десятки и единицы секунд;
- третий байт, mm – десятки и единицы минут;
- четвертый (старший) байт, hh – флаги, десятки и единицы часов (рис. 3.6). Необходимо отметить, что в четвертом байте помимо десятков и единиц часов, размещены также два флага: f2 – выбор 12-часовой (f2 = 1) или 24-часовой (f2 = 0) шкалы времени; f1 – выбор времени суток: *до полудня* (англ. *am*) (f1 = 0) или *после полудня* (англ. *pm*) (f1 = 1). Флаг f1 используется только при работе с 12-часовой шкалой.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
f2	f1	d	d	e	e	e	e	Регистр 04

Рис. 3.6. Структура байта hh: f2 – шкала времени 12-часов (f2 = 1) или 24-часа (f2 = 0); f1 – выбор времени суток *до полудня* (f1 = 0) или *после полудня* (f1 = 1); dd – десятки и eeee – единицы часов в двоично-десятичном коде

В заголовочном файле sdk_base.h подготовлены константы, позволяющие выбрать необходимые режимы работы часов, задать необходимые значения флагов f1 и f2 и сформировать байт hh:

- RTC_12 – выбрать 12-часовую шкалу времени (0x80);
- RTC_24 – выбрать 24-часовую шкалу времени (0x00);
- RTC_AM – задать время суток *до полудня* (0x00);
- RTC_PM – задать время суток *после полудня* (0x40);
- RTC_HH – маска для выделения/исключения флагов f1 и f2 из полученного байта hh (0x3F).

Приведем примеры использования данных констант:

• формирование байта hh в 24-часовой шкале времени для установления 13 часов:

```
hh = (RTC_24 | 0x13); // 13 часов.
```

• формирование байта hh в 12-часовой шкале времени для установления 11 am (11 часов утра) или 11 pm (11 часов вечера):

```
hh = (RTC_12 | RTC_AM | 0x11);
hh = (RTC_12 | RTC_PM | 0x11);
```

• очистка полученного байта hh от флагов f1 и f2:

```
hh_clear = (hh & RTC_HH);
```

• выделение флагов f1 и f2 из полученного байта hh:

```
f2_f1 = (hh & (~RTC_HH));
```

Прототип функции gettime имеет вид:

```
bit gettime(uchar);
```

Аргумент функции задает адрес в области idata для размещения первого принятого байта (миллисекунды). После записи каждого байта в память происходит инкремент адреса, поэтому остальные три байта размещаются в памяти после первого байта подряд по возрастанию адресов. Код возврата равен нулю при успешном приеме четырех байтов, при любой ошибке код возврата будет равен единице.

После приема четырех байтов функция gettime пересылает их в последовательный порт UART. На компьютере должен быть включен монитор T2 в режиме эмуляции терминала (1 term). Использование этого режима позволяет не только демонстрировать принятые байты в окне монитора T2, но и управлять работой функции gettime. Нажатие любой клавиши на клавиатуре компьютера приводит к получению новой порции данных о текущем времени в виде четырех байтов;

puttime служит для демонстрации функции sendblock на примере начальной установки времени в микросхеме часов, входящей в состав стенда. Функция puttime пересылает четыре заранее подготовленных байта из буфера в области idata в микросхему часов, размещая их там в соответствующие внутренние регистры 01...04:

• 01 – сотые доли секунды (первый (младший) байт буфера, ms);
• 02 – секунды (второй байт буфера, ss);
• 03 – минуты (третий байт буфера, mm);
• 04 – флаги и часы (четвертый байт буфера, hh) (рис. 3.6).

Прототип функции puttime имеет вид:

```
bit puttime(uchar);
```

Аргумент функции задает адрес первого байта в области idata, где предварительно размещены четыре байта в двоично-десятичном коде: сотые доли секунды (младший байт), секунды, минуты, флаги

и часы (старший байт). Код возврата равен нулю при успешной передаче четырех байтов, при любой ошибке код возврата будет равен единице. Структура данных в буфере функции `puttime` идентична таковой для функции `gettime`, однако взаимное расположение этих двух буферов в памяти `idata` достаточно произвольное, допускается их полное совмещение.

На рис. 3.7 приведены исходные тексты ассемблерных модулей функций `gettime` и `puttime`. Основу этих функций составляют библиотечные процедуры `getack`, `sendblock` и `receiveblock`. Остальные команды готовят входные параметры для вызова этих процедур:

```
;---------------------
_gettime:
    push acc
    push b
    push 0h
    clr  f0
    mov  a,r7
    mov  r5,a
    mov  r7,#0A0h
    call _getack
    cpl  f0
    jb   f0,get_t1
    mov  r6,#0A0h
    mov  r7,#1h
    mov  r3,#4
    call _receiveblock
 get_t1:
    mov  c,f0
    pop  0h
    pop  b
    pop  acc
    ret
;---------------------
```

```
;---------------------
_puttime:              ; 01.
    push acc           ; 02.
    push b             ; 03.
    push 0h            ; 04.
    clr  f0            ; 05.
    mov  a,r7          ; 06.
    mov  r5,a          ; 07.
    mov  r7,#0A0h      ; 08.
    call _getack       ; 09.
    cpl  f0            ; 10.
    jb   f0,put_t1     ; 11.
    mov  r6,#0A0h      ; 12.
    mov  r7,#1h        ; 13.
    mov  r3,#4         ; 14.
    call _sendblock    ; 15.
 put_t1:               ; 16.
    mov c,f0           ; 17.
    pop  0h            ; 18.
    pop  b             ; 19.
    pop  acc           ; 20.
    ret                ; 21.
;---------------------
```

Рис. 3.7. Исходные тексты ассемблерных модулей функций `gettime` и `puttime`

строки `02...04` — сохранение в используемых регистров;

строка `05` – очистка `f0`, где будет сформирован код возврата;

строки `06...08` – подготовка параметров для вызова библиотечной процедуры `getack`: в регистре `r7` задается адрес первого размещаемого байта в области `idata` (адрес буфера). Сохраняем адрес буфера в `r5`, т. к. в регистр `r7` далее загружаем `I2C`-адрес часов (`OA0h`);

строка `09` – вызов процедуры `getack` для проверки готовности микросхемы часов к обмену данными;

строки `10...11` – анализ кода возврата процедуры `getack`: если был получен `0` (часы не откликаются), то переход к строке `16`, для завершения функции;

строки 12...14 – подготовка входных данных для вызова библиотечной процедуры (**receiveblock** или **sendblock**): I2C-адрес часов (**r6**), адрес первого регистра во внутреннем адресном пространстве часов (**r7**) и ожидаемое количество байтов при обмене данными с микросхемой часов (**r3**);

строка 15 – вызов библиотечной процедуры (**receiveblock** или **sendblock**) для обмена данными между буфером (**idata**) и микросхемой часов;

строка 17 – формирование кода возврата функции;

строки 18...21 – восстановление из стека использованных регистров и возврат из подпрограммы;

putmode предназначена для загрузки одного байта в регистр управления/состояния часов. Регистр имеет адрес **0x0** во внутреннем адресном пространстве ведомого I2C-устройства (микросхемы часов в составе лабораторного стенда SDK) и поддерживает свободный доступ в процессе операций чтения/записи через шину I2C. На рис. 3.8 приведена структура регистра управления/состояния.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Регистр 00

Рис. 3.8. Структура регистра управления/состояния (адрес 0x0)

Назначение битов регистра управления/состояния (адрес **0x0**, исходное состояние **0x0**):

- a_7 – флаг останова счета (**0** – импульсы счета; **1** – останов счета и сброс делителя);
- a_6 – флаг сохранения последнего считывания (**0** – счет; **1** – сохранить и скопировать в регистры-защелки);
- $a_5\_a_4$ – два бита для выбора режима работы (**00** – режим часов на частоте 32.768 кГц; **01** – режим часов на частоте 50 Гц; **10** – режим счетчика событий; **11** – тестовый режим);
- a_3 – флаг маски (**0** – немаскируемое чтение адресов **05...06**; **1** – непосредственное чтение даты и месяца);
- a_2 – бит разрешения сигнала (**0** – сигналы отключены: переключение флагов, регистр управления сигналом отключен (адреса памяти с **08** по **0F** свободны); **1** – включение регистра управления сигналом (ячейка памяти по адресу **08** – регистр управления сигналом));
- a_1 – флаг сигнала (50 % занятости флага минут, если бит разрешения сигнала равен нулю);
- a_0 – флаг таймера (50 % занятости флага секунд, если бит разрешения сигнала равен нулю).

Прототип функции **putmode** имеет следующий вид:

```
bit putmode (uchar);
```

Аргумент функции putmode – байт, предназначенный для загрузки в регистр управления/состояния микросхемы часов.

Возвращаемое значение является флагом завершения процесса:

- 0 – байт успешно загружен в регистр управления/состояния;
- 1 – ведомое устройство I2C не откликается;

getmode предназначена для получения одного байта из регистра управления/состояния (внутренний адрес регистра 00) микросхемы часов в составе лабораторного стенда SDK.

Прототип функции getmode имеет вид:

```
uchar getmode (void);
```

Функция getmode не имеет входных аргументов (void).

Возвращаемое значение – прочитанный байт, т. е. содержимое регистра управления/состояния I2C-устройства микросхемы часов. Если операция чтения закончилась неудачно (ведомое устройство I2C не откликается), то возвращаемое значение будет равно 0xFF.

В заголовочном файле sdk_base.h подготовлены два макроопределения, позволяющие включать (RTC_START) режим счета или останавливать счет со сбросом делителя (RTC_STOP). Рекомендуется останавливать счет (RTC_STOP) перед загрузкой исходных данных в I2C-устройство микросхемы часов, а запуск часов (RTC_START) производить только после загрузки исходных данных в часы.

3.1.5. Аналого-цифровые и цифро-аналоговые преобразователи

В разделе описаны функции библиотеки sdk_base, предназначенные для демонстрации функционирования аналого-цифрового (АЦП) (англ. *analog to digital converter*, ADC) и цифро-аналогового (ЦАП) (англ. *digital to analog converter*, DAC) преобразователей, которые входят в состав стендов типа SDK, являясь встроенным оборудованием микроконтроллеров aduc8xx:

adc_ini служит для загрузки трех байтов управляющей информации в соответствующие регистры управления/состояния: adccon1, adccon2 и adccon3. На рис. 3.9 приведена структура регистров управления/состояния аналого-цифрового преобразователя. Кратко обсудим назначение битов этих регистров. Исчерпывающее описание этих регистров содержится в рекомендуемых руководствах [6–8].

Спецификация регистра adccon1:

- $m_1\,m_0$ – режим электропитания АЦП aduc812 (00 – дежурный; 01 – нормальный; 10 – дежурный при отсутствии цикла преобразования; 11 – холостой при отсутствии цикла преобразования);
- $ck_1\,ck_0$ – настройка коэффициента делителя тактовой частоты;
- $aq_1\,aq_0$ – настройка задержки для переключения коммутатора;
- t2c – разрешение запуска от таймера C/T2;

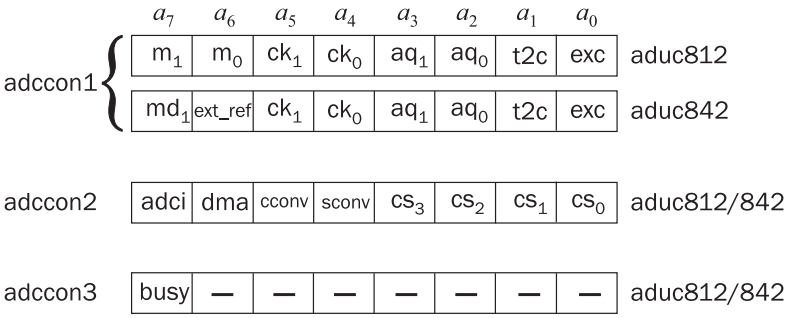

Рис. 3.9. Структура регистров управления/состояния АЦП

- `exc` – разрешение внешнего запуска;
- md_1 – включение электропитания АЦП `aduc842`;
- `ext_ref` – выбор источника опорного напряжения АЦП для микроконтроллера `aduc842` (1 – внешний; 0 – внутренний).

Спецификация регистра `adccon2`:

- `adci` – флаг прерывания;
- `dma` – флаг разрешения режима прямого доступа в память;
- `cconv` – запуск циклического преобразования;
- `sconv` – запуск однократного преобразования: устанавливается программно пользователем, сбрасывается аппаратно при завершении преобразования АЦП;

- cs_3 cs_2 cs_1 cs_0 – выбор входного канала коммутатора:

0	n_2	n_1	n_0	– выбор внешнего канала (0...7);
1	0	0	0	– выбор теплового сенсора АЦП;
1	x	x	x	– зарезервированные комбинации;
1	1	1	1	– останов режима ПДП.

Спецификация регистра `adccon3`: `busy` – бит состояния АЦП.

Остальные биты регистра `adccon3` в микроконтроллере `aduc812` зарезервированы под возможное использование в последующих версиях микроконтроллера [6]. В микроконтроллере `aduc842` некоторые из этих битов заняты под управление калибровочными функциями [7]. Однако в учебных задачах эти функции не используются.

При программировании многих учебных задач бывает достаточно задать только биты a_7 и a_6 в регистре `addcon1`. При этом необходимо помнить, что в микроконтроллерах `aduc812` и `aduc842` назначение этих битов отличается. В `aduc812` выбирается *нормальный режим* электропитания (биты $a_7\,a_6 = 01$), а в `aduc842` выбирается включение электропитания АЦП и запрет *внешниего запуска* (биты $a_7\,a_6 = 10$). В заголовочном файле `sdk_base.h` для такого назначения предусмотрены две соответствующие константы `ADC_CON812` и `ADC_CON842`.

При начальной инициализации допустимо сбросить следующие биты: adccon1.5...adccon1.0, adccon2.7...adccon2.4 и все биты в регистре addcon3.

При программировании АЦП необходимо вдумчиво выбрать источник входного аналогового сигнала (биты adccon2.3...adccon2.0: внешний источник сигнала (каналы 0. . . 7 коммутатора) либо внутренний источник сигнала. В обоих контроллерах (aduc812 и aduc842) в качестве внутреннего источника можно использовать тепловой сенсор (канал 08 коммутатора). В заголовочном файле sdk_base.h для выбора теплового сенсора предусмотрена константа ADC_TSENS. Отметим, что в микроконтроллере aduc842 можно выбрать также другие внутренние источники, например источник опорного напряжения (канал 0C коммутатора).

В простейшем случае запуск однократного преобразования осуществляется программной установкой бита adccon2.4 (sconv), контроль завершения цикла преобразования АЦП может осуществляться по его аппаратному сбросу.

Прототип функции adc_ini имеет следующий вид:

```
void adc_ini (uchar, uchar, uchar);
```

Аргументы функции – три байта для загрузки соответственно в управляющие регистры adccon1, adccon2 и adccon3;

adc_single предназначена для демонстрации работы АЦП в режиме однократного преобразования с программным запуском и контролем завершения преобразования с помощью бита adccon2.4.

Прототип функции adc_single имеет следующий вид:

```
uint adc_single (void);
```

Функция не имеет входных аргументов. Возвращаемое значение – 16-битовый операнд (uint), в старшей тетраде которого размещена информация о номере канала выбранного источника входного аналогового сигнала, три младшие тетрады содержат 12-битовый код, полученный в ходе цикла преобразования. В заголовочном файле подготовлены четыре константы ADC_DAT0...ADC_DAT3, которые можно использовать для выделения отдельных тетрад из возвращаемого значения;

dac_ini предназначена для загрузки управляющей информации в регистр управления daccon, который осуществляет выбор режимов и настройку работы обоих цифро-аналоговых преобразователей. На рис. 3.10 приведена общая структура регистра управления цифро-аналогового преобразователя. Кратко обсудим назначение битов этого регистра. Исчерпывающее описание структуры регистра daccon содержится в рекомендуемых руководствах [6–8]:

• mode – выбор 8-разрядного (1) или 12-разрядного (0) режима работы (для обоих ЦАП устанавливается одинаковый режим работы).

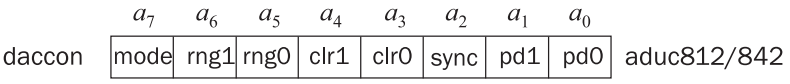

Рис. 3.10. Структура регистра управления ЦАП

Важно отметить, в 8-разрядном режиме необходимо загружать байт в `dacxL`, который после загрузки автоматически сдвигается в верхнюю часть 12-разрядного регистра данных;

• `rng1` и `rng0` – выбор диапазона выходных сигналов для ЦАП1 и ЦАП0: 0–V_{dd} (1) или 0–V_{ref} (0);

• `clr1` и `clr0` – биты очистки выхода: соответствует заданному коду (1), равен нулю (0);

• `sync` – бит синхронизации: при `sync = 0` производится предварительная загрузка регистров `dacxL/H`, при `sync = 1` происходит одновременная модификация ЦАП1 и ЦАП0. Если уже установлен `sync = 1`, то модификация выходов происходит при загрузке младшего байта в `dacxL`.

• `pd1` и `pd0` – включение (1) или выключение (0) электропитания ЦАП1 и ЦАП0 соответственно.

Прототип функции `dac_ini` имеет следующий вид:

```
void dac_ini (uchar);
```

dac_single предназначена для демонстрации работы ЦАП: производит загрузку 12-битного операнда в регистр данных указанного ЦАП и после этого устанавливает бит `sync` для модификации выхода указанного ЦАП.

Прототип функции `dac_single` имеет следующий вид:

```
void dac_single (uint, uchar);
```

Первый аргумент содержит данные для загрузки в 12-битный регистр данных, второй аргумент служит для выбора ЦАП1 (1) или ЦАП0 (0). В заголовочном файле `sdk_base.h` подготовлены две константы для выбора ЦАП по номеру: `DAC_1` и `DAC_0`.

3.2. Вызов библиотечных A-функций из C-программы

3.2.1. Доступ к регистрам ПЛИС

На рис. 3.11 приведен исходный текст C-программы, которая демонстрирует работу четырех обсуждавшихся выше A-функций из библиотеки `sdk_base`: `lputchar`, `lgetchar`, `getkey` и `svdisp`:

`строка 01` содержит указание подключить заголовочный файл, содержащий прототипы библиотечных функций и определяющий некоторые важные константы для работы с этими функциями. Имя заго-

```
//-------------------------------------------------------------
#include "sdk_base.h"                                        // 01.
void main (void)                                             // 02.
{                                                            // 03.
   uchar symbol;                                             // 04.
   uchar *mm = "\x0A0\x0B8\x0B2\x0BB\x0B8o. \                 // 05.
\x0E4y\x0BD\x0BA\x0E5\x0B8\x0B8";                             // 06.
   int i = 0;                                                // 07.
   while(mm[i] != '\0'){                                     // 08.
     lputchar(mm[i++],1);                                    // 09.
     while(((lgetchar()) & LCD_BF) != 0);                    // 10.
   }                                                         // 11.
   while(1){                                                 // 12.
     while((symbol = getkey()) == 0xFF);                     // 13.
     while(getkey() != 0xFF);                                // 14.
     symbol += (symbol > 9)? 55 : 48;                        // 15.
     lputchar(LCD_POS(0x47),0);                              // 16.
     while((lgetchar() & LCD_BF) != 0);                      // 17.
     lputchar(symbol,1);                                     // 18.
     symbol = lgetchar();                                    // 19.
     svdisp(symbol);                                         // 20.
     while((lgetchar() & LCD_BF) != 0);                      // 21.
   }                                                         // 22.
}                                                            // 23.
//-------------------------------------------------------------
```

Рис. 3.11. С-программа, демонстрирующая взаимодействие четырех А-функций из библиотеки `sdk_base`: `lputchar`, `lgetchar`, `getkey` и `svdisp`

ловочного файла набрано в двойных кавычках (`"sdk_base.h"`), поскольку он является пользовательским файлом и размещен в рабочей директории пользователя, а не в стандартной системной директории;

строки 02, 03 – заголовок головной С-программы `main`;

строка 04 содержит декларацию переменной `symbol`;

строки 05, 06 содержат декларацию указателя `mm` на статический массив символов и его инициализацию текстовой строкой `Библио. функции`, которая далее будет выводиться на экран ЖК-дисплея. Кириллические буквы в знакогенераторе ЖК-дисплея имеют нестандартную кодировку, поэтому для инициализации текстовой строки использовали их шестнадцатеричные коды, взятые из таблицы знакогенератора [5], а кириллические символы `'о'`, `'у'` заменили сходными по начертанию латинскими символами. Одиночный обратный слеш `'\'` в конце строки 5 позволяет перенести длинный текст и продолжить его на следующей строке. В конце текстовой строки компилятор автоматически добавляет непечатный нуль-символ `'\0'` с нулевым кодом `NULL`, который служит индикатором конца текстовой строки;

строка 07 содержит декларацию и начальную инициализацию переменной `i`, которая соответствует индексу символьного элемента в текстовом массиве `nm[]`;

строки 08...11 содержат цикл while, который пробегает по всем элементам текстового массива nm[i] пока не достигнет завершающего нуль-символа. Тело цикла содержит две строки 9 и 10, осуществляющих вывод данных и проверку готовности ЖК-дисплея;

строка 09 при каждой итерации цикла while осуществляет вывод на ЖК-дисплей одного текущего элемента текстового массива. Для вывода используется функция lputchar в режиме вывода данных. При выводе символа осуществляется не только его отображение на экране ЖК-дисплея, но и автоматическое смещение курсора вправо на одно знакоместо. Постфиксная запись инкремента nm[i++] означает, что сначала производится чтение элемента массива и передача его в качестве аргумента функции lputchar(nm[i]), лишь после этого происходит инкремент индекса массива: i = i + 1;

строка 10 содержит вложенный цикл while, в поле условия которого осуществляется периодическое чтение ЖК-дисплея и анализ бита готовности (BF) до тех пор пока не будет достигнуть состояние BF = 0. В теле этого цикла не производится никаких действий;

строки 12...22 содержат бесконечный цикл while, в теле которого происходит опрос клавиатуры и ЖК-дисплея, анализ возвращаемых кодов и отображение информации;

строка 13 содержит вложенный цикл while, в поле условия которого осуществляется периодический опрос клавиатуры пока не будет нажата какая-либо клавиша. Код нажатой клавиши сохраняется в переменной symbol. В теле обсуждаемого цикла не совершается никаких дополнительных действий;

строка 14 содержит вложенный цикл while, в поле условия которого осуществляется периодический опрос клавиатуры, пока не будет отпущена (отжата) ранее нажатая клавиша. В теле цикла не совершается никаких дополнительных действий;

строка 15 содержит кодировщик двоичных чисел 0x0...0xF в HEX-коды, т. е. в текстовые символы (0...9, A...F), соответствующие шестнадцатеричному представлению этих двоичных чисел. При анализе алгоритма работы преобразователя кодов необходимо учесть, что в десятичном представлении ASCII-коды этих символов расположены в диапазонах 48...57 (0...9) и 65...70 (A...F);

строка 16. Функция lputchar в режиме команд осуществляет расчет управляющего кода при помощи макрофункцией LCD_POS и перемещение курсора на седьмое знакоместо нижней строки;

строка 17 содержит вложенный цикл while, в поле условия которого осуществляется периодическое чтение ЖК-дисплея и анализ бита готовности (BF) до тех пор, пока не будет достигнуто состояние BF = 0. В теле этого цикла не производится никаких действий;

строка 18. Функция **lputchar** в режиме данных выводит переменную **symbol** на экран дисплея в ранее выбранное знакоместо;

строка 19. Функция **lgetchar** читает выходной порт ЖК-дисплея и сохраняет его в переменной **symbol**;

строка 20. Функция **svdisp** выводит переменную **symbol** на линейку светодиодных индикаторов;

строка 21 содержит вложенный цикл **while**, в поле условия которого осуществляется периодическое чтение ЖК-дисплея и анализ бита готовности (**BF**) до тех пор, пока не будет достигнуто состояние **BF = 0**. В теле этого цикла не производится никаких действий;

строки 22 и 23 завершают цикл **while**, начатый в строке 12, и программу **main** соответственно.

3.2.2. Доступ к внешнему параллельному порту

На рис. 3.12 приведен исходный текст ассемблерного модуля для доступа к внешнему параллельному порту ПЛИС. Доступ осуществляется при помощи библиотечных А-функций **get_lpt** и **put_lpt**.

```
//-------------------------------------------------------------
#include "sdk_base.h"
void main (void)
{
   uchar symbol;
   uchar *mm = "\x0A0\x0B8\x0B2\x0BB\x0B8o. \           // 05.
\x0E4y\x0BD\x0BA\x0E5\x0B8\x0B8";                       // 06.
   int i = 0;
   while(mm[i] != '\0'){
     lputchar(mm[i++],1);
     while((lgetchar() & LCD_BF) != 0);
   }
   while(1){
     i = get_lpt();
     symbol = (char) 0xFF & i;
     svdisp(symbol);
   }
}
//-------------------------------------------------------------
```

Рис. 3.12. С-программа, демонстрирующая совместную работу А-функций из библиотеки **sdk_base**: **get_lpt**, **put_lpt**

3.2.3. Доступ к последовательному порту UART

На рис. 3.13 приведен исходный текст С-программы, которая демонстрирует работу трех обсуждавшихся выше А-функций из библиотеки **sdk_base** (**uart_ini**, **get_uart** и **put_uart**):

```
//-------------------------------------------------------------
#include "sdk_base.h"                                        // 01.
void main (void)                                             // 02.
{                                                            // 03.
   uchar symbol;                                             // 04.
   uchar *mm = "\x0A0\x0B8\x0B2\x0BB\x0B8o. \                 // 05.
\x0E4y\x0BD\x0BA\x0E5\x0B8\x0B8";                             // 06.
   int i = 0;                                                // 07.
   while(mm[i] != '\0'){                                     // 08.
     lputchar(mm[i++],1);                                    // 09.
     while(((lgetchar()) & LCD_BF) != 0);                    // 10.
   }                                                         // 11.
   uart_ini(ADUC842);                                        // 12.
   while(1){                                                 // 13.
     symbol = get_uart();                                    // 14.
     put_uart(symbol);                                       // 15.
     svdisp(symbol);                                         // 16.
     lputchar(LCD_POS(0x47),0);                              // 17.
     while((lgetchar() & LCD_BF) != 0);                      // 18.
     lputchar(symbol,1);                                     // 19.
     while((lgetchar() & LCD_BF) != 0);                      // 20.
   }                                                         // 21.
}                                                            // 22.
//-------------------------------------------------------------
```

Рис. 3.13. С-программа, иллюстрирующая совместную работу трех А-функций из библиотеки sdk_base: uart_ini, get_uart и put_uart

строки 01...11 идентичны таковым для С-программы, которая детально обсуждалась выше (см. рис. 3.11);

строка 12 содержит вызов функции для начальной инициализации контроллера последовательного порта aduc842;

строки 13...21 содержат бесконечный цикл while, в теле которого происходит опрос последовательного порта UART, ожидание прихода байта и сохранение его в переменной symbol (строка 14), отправка принятого байта обратно в линию (строка 15), отображение байта на светодиодных индикаторах (строка 16) и экране ЖК-дисплея в заданной позиции (строки 17...20). Каждый вызов функции lputchar для вывода данных на ЖК-дисплей сопровождается вызовом функции lgetchar, осуществляющей контроль флага готовности BF для синхронизации с интерфейсом ЖК-дисплея.

Для демонстрации программы (рис. 3.13) необходимо лабораторный стенд подключить к последовательному порту компьютера, используя штатный шлейф (плоский кабель). На компьютере следует запустить монитор T2 в режиме эмуляции терминала (0 term). В таком режиме все символы, набираемые на клавиатуре компьютера, будут передаваться по последовательному каналу на стенд. Все принятые байты далее подлежат обработке программой, алгоритм которой описан выше при обсуждении С-программы (рис. 3.13).

3.2.4. Доступ к последовательному порту I2C

На рис. 3.14 приведен исходный текст C-программы, которая демонстрирует работу двух обсуждавшихся выше A-функций из библиотеки sdk_base: gettime, puttime, getmode, putmode и getuart:

```
//-----------------------------------------------------
#define BUF 0x40      // Адрес буфера в области idata.   01.
#define BUF_SIZE 4            // Размер буфера, байт.   02.
#include "sdk_base.h"                                  // 03.
void main (void)                                       // 04.
{                                                      // 05.
   uchar idata *ptr, *var;                             // 06.
   int i;                                              // 07.
   ptr = (uchar idata *) BUF;                          // 08.
   uart_ini(ADUC842);                                  // 09.
   putmode (RTC_STOP);      // Останов счетчика часов.   10.
   var = ptr;                                          // 11.
   *var++ = 0x00;               // Сотые доли секунды.   12.
   *var++ = 0x50;                     //      Секунды.   13.
   *var++ = 0x59;                     //       Минуты.   14.
   *var   = (0x12 | RTC_12 | RTC_PM); // Часы и флаги.   15.
   puttime(BUF);                                       // 16.
   get_uart();      // Ожидание клавиатуры компьютера.   17.
   svdisp(getmode());// Вывод полученного байта на SV.   18.
   putmode (RTC_START);        // Пуск счетчика часов.   19.
   while(1){                                           // 20.
      var = ptr;                                       // 21.
      get_uart();   // Ожидание клавиатуры компьютера.   22.
      gettime(BUF);                                    // 23.
      for(i=0;i < BUF_SIZE - 1;i++) {                  // 24.
         put_uart(*var++);                             // 25.
      }                                                // 26.
      put_uart((*var) & RTC_HH); // Очистка флагов hh.   27.
   }                                                   // 28.
}                                                      // 29.
//-----------------------------------------------------
```

Рис. 3.14. C-программа, демонстрирующая совместную работу двух A-функций из библиотеки sdk_base: gettime и puttime

строки 01, 02 определяют константы, содержащие адрес буфера в области idata и его размер в байтах;

строка 03 – подключение заголовочного файла библиотеки;

строки 04, 05 соответствуют началу головной функции main;

строка 06 – объявление указателей ptr и var;

строка 07 – объявление переменной i, выполняющей функцию счетчика байтов в буфере BUF;

строка 08 – преобразование адреса буфера в указатель;

строка 09 – инициализация последовательного порта UART;

строка 10 – останов счетчика часов и сброс делителя;

строка 11 – инициализация переменной **var** значением **ptr**;

строки 12...14 – размещение в буфере в области **idata** трех ДДК-байтов (**ms/ss/mm**), соответствующих текущему времени;

строка 15 – добавление текущего значения часов и двух флагов для назначения 12-часовой шкалы времени и задания времени суток *после полудня* (pm);

строка 16 – начальная установка времени в микросхеме часов;

строка 17 – ожидание прихода сигнала с клавиатуры компьютера (нажатие на любую клавишу);

строка 18 – получение байта из регистра управления/состояния часов и отображение его на светодиодных индикаторах;

строка 19 – пуск счетчика часов;

строки 20...28 – бесконечный цикл **while**, в теле которого выполняется инициализация переменной **var** (строка 21), происходит ожидание прихода какого-либо байта на вход приемника последовательного порта (строка 22), получение четырех байтов текущего времени (**ms/ss/mm/hh**) и размещение принятых байтов в буфере (строка 23), цикл **for**, осуществляющий пересылку трех принятых байтов (**ms/ss/mm**) из буфера в последовательный порт **UART** (строки 24...26), очистка байта **hh** от флагов в двух старших битах и пересылка значения часов в последовательный порт **UART** (строка 27).

Для работы этой программы на компьютере должен быть запущен монитор **T2** в режиме эмуляции терминала (**1 term**).

На рис. 3.15 показан исходный текст С-программы, которая демонстрирует работу двух обсуждавшихся выше А-функций из библиотеки **sdk_base**: **sendblock** и **receiveblock**. Для пояснения интерфейса доступа к этим функциям используются две вспомогательные С-функции **gettime_c** и **puttime_c**, которые воспроизводят работу соответствующих библиотечных А-функций **gettime** и **puttime**:

строки 01...31 почти идентичны головной программе **main**, приведенной на рис. 3.14. Имеют место только два отличия. Во-первых, добавлены прототипы вспомогательных С-функций (строки 04, 05). Во-вторых, вместо библиотечных функций **gettime** и **puttime**, показанных на рис. 3.14, вызываются соответствующие вспомогательные С-функции **gettime_c** и **puttime_c** (строки 18, 25);

строки 32...42 – исходный текст С-функции **gettime_c**;

строка 34 – декларация и очистка битовой переменной **flag_f0**, в которой будет далее готовиться код возврата;

строка 35 – декларация переменной **ii** и загрузка ее **I2C**-адресом микросхемы часов;

```
#define BUF 0x40      // Адрес буфера в области idata.   01.
#define BUF_SIZE 4             // Размер буфера, байт.   02.
#include "sdk_base.h"                                   // 03.
bit gettime_c(uchar);                                   // 04.
bit puttime_c(uchar);                                   // 05.
void main (void)                                        // 06.
{                                                       // 07.
   uchar idata *ptr, *var;                              // 08.
   int i;                                               // 09.
   ptr = (uchar idata *) BUF;                           // 10.
   uart_ini(ADUC842);                                   // 11.
   putmode (RTC_STOP);                                  // 12.
   var = ptr;                                           // 13.
   *var++ = 0x00;               // Сотые доли секунды.   14.
   *var++ = 0x50;                     //      Секунды.   15.
   *var++ = 0x59;                     //       Минуты.   16.
   *var   = (0x12 | RTC_12 | RTC_PM); // Часы + флаги.   17.
   puttime_c(BUF);                                      // 18.
   get_uart();                                          // 19.
   svdisp(getmode());                                   // 20.
   putmode (RTC_START);                                 // 21.
   while(1){                                            // 22.
      var = ptr;                                        // 23.
      get_uart();                                       // 24.
      gettime_c(BUF);                                   // 25.
      for(i=0;i<BUF_SIZE - 1;i++) {                     // 26.
         put_uart(*var++);                              // 27.
      }                                                 // 28.
      put_uart((*var) & RTC_HH);                        // 29.
   }                                                    // 30.
}                                                       // 31.
bit gettime_c(uchar buffer)                             // 32.
{                                                       // 33.
   bit flag_f0 = 0;                                     // 34.
   int ii = 0xA0;                  // I2C адрес часов.   35.
   flag_f0 = ~(getack (ii));                            // 36.
   if(flag_f0 != 1) {                                   // 37.
      ii = (ii << 8) | 0x1;                             // 38.
      flag_f0 = receiveblock (ii, buffer, BUF_SIZE);    // 39.
   }                                                    // 40.
   return(flag_f0);                                     // 41.
}                                                       // 42.
bit puttime_c (uchar buffer)                            // 43.
{                                                       // 44.
   bit flag_f0 = 0;                                     // 45.
   int ii = 0xA0;                  // I2C адрес часов.   46.
   flag_f0 = ~(getack(0xA0));                           // 47.
   if(flag_f0 != 1) {                                   // 48.
      ii = (ii << 8) | 0x1;                             // 49.
      flag_f0 = sendblock (ii, buffer, BUF_SIZE);       // 50.
   }                                                    // 51.
   return(flag_f0);                                     // 52.
}                                                       // 53.
```

Рис. 3.15. C-программа, демонстрирующая совместную работу двух A-функций из библиотеки `sdk_base`: `sendblock` и `receiveblock`

строка 36 – вызов функции getack для проверки готовности микросхемы часов к обмену данными, получение кода возврата (1 – часы готовы, 0 – часы не откликаются) и его инверсия для последующего анализа;

строка 37 – анализ инвертированного кода возврата getack. Если микросхема часов не откликнулась, то произойдет переход к точке выхода из С-функции;

строка 38 – подготовка двухбайтового первого аргумента функции receiveblock: адреса часов (H) и регистра в часах (L);

строка 39 – вызов библиотечной процедуры receiveblock для получения из микросхемы часов четырех ДДК-байтов и размещения их в буфере;

строка 41 – формирование кода и возврат из С-функции;

строки 43...53 – исходный текст С-функции puttime_c;

строка 45 – декларация и очистка битовой переменной flag_f0, в которой будет далее готовиться код возврата;

строка 46 – декларация переменной ii и загрузка ее I2C-адресом микросхемы часов;

строка 47 – вызов функции getack для проверки готовности микросхемы часов к обмену данными, получение кода возврата (1 – часы готовы, 0 – часы не откликаются) и его инверсия для последующего анализа;

строка 48 – анализ инвертированного кода возврата getack. Если микросхема часов не откликнулась, то произойдет переход к точке выхода из С-функции;

строка 49 – подготовка двухбайтового первого аргумента функции sendblock: адрес часов (H) и адрес регистра в часах (L);

строка 50 – вызов библиотечной процедуры sendblock для отправки в микросхему часов четырех ДДК-байтов, предварительно подготовленная в буфере;

строка 52 – формирование кода и возврат из С-функции.

3.2.5. Доступ к АЦП и ЦАП

На рис. 3.16 приведена программа, демонстрирующая работу АЦП и осуществляющая следующие действия: начальная инициализация АЦП, однократный запуск преобразования сигнала от внутреннего температурного сенсора, извлечение из выходной информации АЦП четырех тетрад и их вывод на ЖК-дисплей, опрос клавиатуры стенда для перехода к следующему преобразованию с индикацией события на светодиодном индикаторе. Рассмотрим коды этой программы:

```
//-------------------------------------------------
#include "sdk_base.h"                                   // 01.
void adc2lcd(uint, uchar);                              // 02.
void main(void)                                         // 03.
{                                                       // 04.
   uint adcdigit;                                       // 05.
   adc_ini (ADC_CON842, ADC_TSENS, 0x0);                // 06.
   while(1) {                                           // 07.
     svdisp(0x01);                                      // 08.
     adcdigit = adc_single();                           // 09.
     lputchar(LCD_POS(0x45),0);                         // 10.
     while((lgetchar() & LCD_BF) != 0);                 // 11.
     adc2lcd (adcdigit, ADC_DAT3);                      // 12.
     lputchar(':',1);                                   // 13.
     while((lgetchar() & LCD_BF) != 0);                 // 14.
     adc2lcd (adcdigit, ADC_DAT2);                      // 15.
     adc2lcd (adcdigit, ADC_DAT1);                      // 16.
     adc2lcd (adcdigit, ADC_DAT0);                      // 17.
     while((getkey()) == 0xFF);                         // 18.
     svdisp(0x0);                                       // 19.
     while(getkey() != 0xFF);                           // 20.
   }                                                    // 21.
}                                                       // 22.
void adc2lcd(uint word, uchar number)                   // 23.
{                                                       // 24.
   uchar ch;                                            // 25.
   ch = (uchar) (word >> number);                       // 26.
   ch = ch & 0x0F;                                      // 27.
   ch += (ch > 9)? 55 : 48;                             // 28.
   lputchar(ch,1);                                      // 29.
   while((lgetchar() & LCD_BF) != 0);                   // 30.
   return;                                              // 31.
}                                                       // 32.
//-------------------------------------------------
```

Рис. 3.16. С-программа, демонстрирующая работу двух А-функций из библиотеки `sdk_base`: `adc_ini` и `adc_single`

строка 01, 02 – подключение заголовочного файла библиотеки и объявление прототипа вспомогательной функции;

строка 03 соответствует началу головной программы;

строка 05 – объявление вспомогательной переменной для фиксации результата;

строка 06 – начальная инициализация АЦП;

строки 7...21 – бесконечный цикл `while`, в котором выполняются следующие действия: включение светодиодного индикатора (08), цикл однократного преобразования АЦП и взвращение двухбайтового результата – номер канала и 12-битовый код оцифровки (09), установление курсора ЖК-дисплея в середину нижней строки (10, 11), извлечение старшей тетрады результата (номер канала) и ее вывод

на ЖК-дисплей (12), вывод двоеточия для отделения номера канала от последующего кода оцифровки (13, 14), извлечение и вывод на ЖК-дисплей трех тетрад кода оцифровки в шестнадцатеричном представлении (15...17), опрос клавиатуры и управление светодиодным индикатором (18...20), завершение текущей итерации цикла и переход к новой итерации (21);

строки 23...32 – вспомогательная функция adc2lcd, которая извлекает из двухбайтового результата заданную тетраду, кодирует ее в один символ шестнадцатеричного представления и выводит его на ЖК-дисплей. Для этого осуществляются следующие действия: объявление вспомогательной байтовой переменной ch (25), выделение и фиксация в переменной ch заданной тетрады путем сдвига в двухбайтовом результате на заданное количество битов, преобразование типов и очистка старшей тетрады в байтовой переменной ch (26, 27), преобразование выделенной тетрады в ascii-код соответствующей шестнадцатеричной цифры (28), вывод полученного символа на ЖК-дисплей (29, 30), возврат из функции (31).

На рис. 3.17 показана С-программа для демонстрации работы библиотечных функций dac_ini и dac_single. В строке 02 задается номер ЦАП (1 или 0); в строке 03 автоматически выбирается одно из двух подготовленных управляющих слов; в строке 06 происходит инициализация ЦАП. Далее идет бесконечный цикл (7...14), в котором выводится код для напряжения на выходе примерно 730 мВ; ожидается нажатие любой клавиши стенда и сбрасывается выходной сигнал; после отпускания клавиши цикл повторяется. Нажатие и отпускание клавиши сопровождается светодиодной индикацией.

```
//-----------------------------------------------------
#include "sdk_base.h"                                   // 01.
#define DAC_NUM DAC_1                                   // 02.
#define DAC_MODE ((DAC_NUM)? 0x52 : 0x29)               // 03.
void main (void)                                        // 04.
{                                                       // 05.
   dac_ini (DAC_MODE);                                  // 06.
   while(1) {                                           // 07.
     svdisp(0x01);                                      // 08.
     dac_single (0x250,DAC_NUM); // U = 730 mV;           09.
     while((getkey()) == 0xFF);                         // 10.
     dac_single (0x0,DAC_NUM);   // U = 0 mV;           // 11.
     svdisp(0x0);                                       // 12.
     while(getkey() != 0xFF);                           // 13.
   }                                                    // 14.
}                                                       // 15.
//-----------------------------------------------------
```

Рис. 3.17. С-программа, демонстрирующая совместную работу двух А-функций из библиотеки sdk_base: dac_ini и dac_single

4. Требования к оформлению С-программ

Возможность последующей модификации и дальнейшего развития программы после завершения этапов ее разработки, отладки и тестирования определяется многими факторами, одним из которых является следование определенным правилам оформления и комментирования текста программы. Исходный текст программы должен быть информативным, т. е. он должен содержать достаточно информации, чтобы при последующей его модификации программист мог легко понять существенные детали алгоритма и особенности его реализации. Действительно, если исходный текст программы был написан без соблюдения правил оформления и комментирования, то он будет представлять собой просто плохо читаемую и непонятную «мешанину» операторов и знаков препинания. Вносить изменения в такой малоинформативный исходный текст будет очень сложно не только стороннему программисту, но даже самому автору программы, т. к. сложности модификации программы существенно возрастают по прошествии времени даже для автора, а тем более для стороннего программиста, вынужденного работать с исходным текстом чужой программы.

В настоящей главе рассматриваются некоторые общепринятые требования и рекомендации, позволяющие читателю выработать хороший стиль оформления исходного текста программы и повысить его информативность. Данные требования обсуждаются на примере языка C/C^{++}, однако сформулированные правила применимы к любому алгоритмическому языку, например к языку Turbo Pascal [9].

4.1. Соглашения по идентификаторам

Начинающие программисты, еще не усвоившие общепринятый (хороший) стиль оформления исходного текста, зачастую следуют ошибочной практике назначения всем переменным каких-либо совершенно неинформативных однобуквенных имен, например, `m`, `n`, `a`, `s`, `p` и т. п. Во-первых, в этом случае теряется сам смысл понятия имени, а во-вторых, очень высока вероятность того, что назначенное пользователем однобуквенное имя случайно совпадет с зарезервированным словом используемого языка программирования. По этой причине однобуквенные имена принято давать только индексам, например, `i`, `j`, `k`, `m`, `n` и т. п. Исключением из этого правила является случай, когда функция или процедура содержит очень небольшое количество переменных (как правило, не более трех). В этом случае смысл переменных может быть хорошо понятен из контекста или комментариев.

Всем основным переменным, которые имеют принципиальное значение для исполняемого алгоритма и не классифицируются как вспомогательные, необходимо назначить достаточно информативные имена. Назначенные имена должны в той или иной мере пояснять функциональное назначение этих переменных, например, `filename` (имя файла), `int_vector` (целочисленный вектор), `size` (размер), `sum` (сумма), `maximum` (максимум) и т. п. Если программист испытывает затруднения с подбором английских слов, то можно использовать латинскую транслитерацию, т. е. звуковые аналоги русских слов, записанные латинскими буквами, например, `nazv_faila`, `razmer`, `summa` и т. п. Это половинчатое решение, которое не полностью соответствует правилам «хорошего» стиля оформления исходного текста программы, однако оно намного предпочтительнее использования совершенно бессмысленных имен.

Более крупным объектам (например, функциям, классам и т. п.) также необходимо назначать достаточно информативные имена, которые для увеличения смысловой нагрузки могут оказаться достаточно длинными. Существуют два альтернативных способа выбора имен для крупных объектов.

В первом случае имя объекта составляют из нескольких компонентов, каждый из которых начинается с прописной (англ. *initial capitals*) буквы, например `ObjectList`, `ArcSet` и т. п. При этом имена функций рекомендуется начинать с глагола, например, `GetPersonName`, `SetNewDate` и т. п.

Во втором случае компоненты имени могут начинаться со строчной буквы (англ. *lower case*), но для выделения компонентов имени объекта рекомендуется использовать символ подчеркивания, например `add_record`, `copy_object` и т. п.

Сказанное выше в полной мере относится к назначению имен файлов с программами, которые также должны нести обязательную смысловую нагрузку, поясняющую их «содержимое». Так, например, заголовочный файл, содержащий описание класса `vector`, вполне уместно назвать `vector.h` или `vector_description.h`, а файл с описанием методов этого класса можно назвать `vector.cpp` или `vector_implementation.cpp`. Нет никакой необходимости искусственно укорачивать или упрощать имена файлов, поскольку все современные операционные системы (`Windows`, `Linux` и т.п.) предоставляют полную поддержку для использования длинных имен файлов. При разработке крупного программного продукта количество файлов с исходными текстами программных модулей может исчисляться многими сотнями. При этом даже их автору совершенно невозможно запомнить огромный объем информации, поясняющий содержимое каждого файла. Отсюда следует необходимость назначения длинных и информативных имен файлов, т. е. само длинное имя файла должно содержать достаточно информации для понимания функциональ-

ного назначения и содержимого этого файла. При большом количестве файлов появляется необходимость их рационального хранения, например в виде дерева. Для приведенного выше примера файлы `vector\vector.h` и `vector\vector.cpp` могут быть размещены в отдельной директории (каталоге) с именем `vector`.

4.1.1. Подбор идентификаторов

Рассмотрим несколько примеров удачных или неудачных вариантов подбора имен переменных, т. е. *идентификаторов*.

При выборе идентификатора следует стремиться к тому, чтобы он нес максимальную семантическую нагрузку, т. е. был максимально информативным и читаемым. Ниже приведены примеры удачных идентификаторов:

```
const float Eps = 0.0001;     // Точность.
unsigned short Sum;           // Сумма.
unsigned char Message[ 20 ]; // Сообщение.
```

Неудачными можно считать следующие идентификаторы, которые малоинформативны, хотя и не являются однобуквенными:

```
const float UU = 0.0001;      // Точность.
unsigned short Kk;            // Сумма.
unsigned char Zz[ 20 ];       // Сообщение.
```

Идентификаторы следует выбирать из слов английского языка, которые характеризуют функциональное назначение переменной:

```
// Выдает звуковой сигнал заданной частоты и длительности.
void Beep( unsigned short Hertz, unsigned short MSec );
// Выдает True (1), если файл с именем FName существует.
unsigned char ExistFile( unsigned char* FName );
// Признак окончания работы с программой.
unsigned char Done;
// Размеры изделия (ширина, высота).
unsigned short Width, Height;
```

Приемлемыми, но не очень удачными следует считать идентификаторы, полученные транслитерацией русских слов:

```
// Выдает звуковой сигнал заданной частоты и длительности.
void Zvuk(unsigned short Chast, unsigned short Dlit );
// Выдает True (1), если файл с именем Im существует.
unsigned char EstFile(unsigned char* Im );
// Признак окончания работы с программой.
unsigned char Konec;
// Размеры изделия (ширина, высота).
unsigned short Shirina, Vysota;
```

4.1.2. Написание идентификаторов

При оформлении исходного текста программы рекомендуется использовать один из двух основных способов написания идентификаторов. Можно использовать любой из них, но необходимо выбрать один способ и придерживаться его при написании программы:

1) при составлении любого идентификатора каждое слово, входящее в его состав, начинается с заглавной буквы, тогда как остальные символы пишутся строчными буквами:

```
float NextX, LastX; // Следующая и предыдущая итерация.
char BeepOnError;   // Подавать ли звуковой сигнал при
                    // неправильном вводе пользователя?
unsigned char FileName[ 20 ];
// Стандартная функция модуля Graph; выдает описание
// ошибки использования графики по ее коду.
unsigned char* GraphErrorMsg( short ErrCode );
```

2) любой идентификатор пишется строчными буквами, но между словами, входящими в состав идентификатора, помещают разделительный символ подчеркивания '_':

```
float next_x, last_x; // Следующая и предыдущая итерация.
char beep_on_error;   // Подавать ли звуковой сигнал при
                      // неправильном вводе пользователя?
unsigned char file_name[ 20 ];
// Стандартная функция модуля Graph; выдает описание
// ошибки использования графики по ее коду.
unsigned char* graph_error_msg( short err_code );     // .
```

4.2. Соглашения по самодокументированности С-программ

4.2.1. Комментарии

Написание содержательного комментария может потребовать значительных временных затрат, которые однако многократно окупятся при необходимости последующей модификации и развитии программы. Ниже приводятся рекомендации по комментированию:

1) комментарии в теле программы следует писать на русском языке и по существу так, чтобы программист, не участвовавший в разработке программы (но имеющий опыт работы на языке Си), мог без особого труда разобраться в логике программы, и при необходимости сопровождать этот программный продукт;

2) рекомендуется комментарии к программе писать после символов `//`, а парные операторы `/*` и `*/` использовать только при отладке программы как «заглушки» участков программного кода.

В то же время не следует впадать в другую крайность – комментировать все подряд, включая самоочевидные действия, как это показано в следующем примере неудачного комментирования:

```
size = 10; //Присвоить size значение 10
for( i=0; i<size; i++) //Цикл по i от 0 до size
{
. . .
}
```

Подобного рода комментарии только засоряют исходный текст.

С другой стороны, в обязательном порядке необходимо комментировать следующие принципиально важные объекты:

1) заголовок файла для описания содержимого этого файла;

2) заголовок функции для пояснения назначения ее аргументов и смысла самой функции;

3) вводимые/возвращаемые переменные и структуры данных;

4) Основные этапы и особенности реализуемых алгоритмов;

5) любые места, которые трудны для быстрого понимания, в особенности использование различных программных «трюков» (англ. *trick*) и нестандартных приемов.

4.2.2. Спецификация функций

При оформлении исходного текста каждую пользовательскую функцию или процедуру следует дополнить ее описанием, которое обычно выполняется в виде комментариев, содержащих следующие специфицируемые данные [9]:

1) функциональное назначение процедуры;

2) описание семантики параметров-значений (т. е. параметров, передаваемых по их значению);

3) описание семантики параметров-переменных (т. е. параметров, передаваемых по указателю);

4) описание семантики возвращаемого значения.

Ниже приведен пример описания функции:

```
//////////////////////Gauss//////////////////////////////
// Решение системы линейных алгебраических уравнений
// методом Гаусса.
// Вход:
// A - матрица коэффициентов системы;
// B - столбец свободных членов системы;
// Eps - точность вычислений.
// Выход:
// X - вектор решения;
// HasSolution - флаг, устанавливаемый в True, если
// решение системы существует, и в False
```

```
// во всех остальных случаях;
// NumOfRoots - число корней в решении системы, может
// принимать значения:
// 0 - если решение системы не
// существует,
// MaxN - если решение системы
// существует и единственно,
// MaxInt - если существует бесконечное
// множество решений;
// Det - значение определителя матрицы A;
// AForReverse - нижняя треугольная матрица,
// полученная из A в результате
// выполнения прямого хода алгоритма
// Гаусса;
// BForReverse - столбец свободных членов, полученный
// из B в результате выполнения
// прямого хода алгоритма Гаусса.
// Результат: 0 - успешно,1 - ошибка.
////////////////////////////////////////////////////////////

unsigned char Gauss(Matrix A, Vector B, float Eps, Vector* X, \
char* HasSolution, short* NumOfRoots, float* Det, \
Matrix* AForReverse, Vector* BForReverse )
{
    ...
    return 0;
    ...
    return 1;
}
```

Если в описываемой функции реализован какой-либо вычислительный метод (например, оптимизация данных методом *Левенберга-Марквардта*, поиск максимума функции методом *условного градиента* и т. п.), то рекомендуется в теле функции разместить комментарий с кратким описанием особенностей используемого метода либо указать ссылку на литературный источник, где описан рассматриваемый вычислительный метод [9].

4.2.3. Спецификация программного файла или модуля

Каждый файл с исходным текстом программы или модуля должен предваряться спецификацией в виде комментариев, содержащих следующую информацию [9]: условный код и название проекта; назначение, имя и версию файла; фамилию автора; описание модуля; историю изменений модуля. Ниже приведен пример спецификации файла:

```
/*-------------------------------------------------------------
Проект: ABC-1.0
Название: Математические расчеты
Файл: primes.c
Версия: 1.0.2
Автор: Иванов И.И.
```

```
Описание: Подсчет количества простых чисел.
Изменения:
-------------------------------------------------------------
N Дата Версия Автор Описание
-------------------------------------------------------------
1 01.03.07 1.0.1 Иванов И.И. Расчет в промежутке [1..200].
2 05.07.07 1.0.2 Иванов И.И. Расчет в заданном промежутке .
-------------------------------------------------------------*/
```

Следом за спецификацией программного файла рекомендуется разместить комментарии с подробными указаниями по запуску программы и работе с ней, а также ссылку на источник, который был использован при составлении программы (модуля). Эти указания по использованию модуля предназначены для сторонних программистов.

4.3. Соглашения по читаемости программ

4.3.1. Длина строк программного текста

Длина строк программы не должна превышать стандартную ширину экрана, обычно это 80 символов.

Данное соглашение является скорее рекомендацией, которой, однако, следует придерживаться при оформлении исходного текста программы. Дело в том, что технические возможности современных компиляторов позволяют использовать строки много большей длины. При этом конкретное значение этой предельной длины может быть различным для разных компиляторов. В качестве примера приведем данные из технического руководства компилятора Keil C51: максимальная длина строки в исходном тексте программы не должна превышать 2000 символов [4, с. 371]. Однако любая попытка использования длинных строк, выходящих за пределы экрана, существенно понизит читаемость программы.

4.3.2. Количество операторов в строке

Особенности человеческого восприятия информации требуют, чтобы в каждой строке исходного текста программы размещался только один оператор (табл. 4.1). Это не только улучшает читаемость исходного текста, но и существенно облегчает пошаговую отладку в символьных отладчиках. Конечно, длина программы при этом увеличится, но все реальные программы и без того настолько длинные, что добавление еще нескольких страниц (или даже десятков страниц) существенно не изменит общую картину. Выигрыш в информативности исходного текста с избытком компенсирует увеличение длины.

Два оператора в строке вполне допустимы, если второй подчинен первому, причем является единственным подчиненным, например:

Таблица 4.1

Количество операторов в строке

Строка	Неправильно	Правильно

1.
2.
3.
	`int *ptr; ptr = new int [100];`
	`ptr [0] = 0;`
		`int *ptr;`
		`ptr = new int [100];`
		`ptr [0] = 0;`

```
for( k = 0; k < size; k++ ) m [k] = 0;
```

Размещение двух и более операторов в строке бывает допустимо и даже желательно, если это позволяет подчеркнуть некую систему в локальной последовательности операторов, например:

```
x1 = Tr1 [0];    y1 = Tr1 [1];    z1 = Tr1 [2];
x2 = Tr2 [0];    y2 = Tr2 [1];    z2 = Tr2 [2];
x3 = Tr3 [0];    y3 = Tr3 [1];    z3 = Tr3 [2];
```

4.3.3. Отступы

Ключевым методом, обеспечивающим быстрое визуальное восприятие (читаемость) исходного текста программы, является правильное использование отступов, которые должны зрительно демонстрировать иерархическую структуру операторов. При этом директивы препроцессора (`#include`, `#define` и т. п.), описания классов, структур, типов, глобальных данных, прототипы и описания функций всегда имеют наивысший приоритет, т. е. начинаются с крайней левой позиции, например:

```
#include <stdio.h>
#define NAME_SIZE 256
int main()
{
.....................
}
```

Размер отступа не должен быть ни слишком мал, ни слишком велик. Оптимальная величина составляет 2–5 пробелов. Один раз выбрав величину отступа, нужно ей придерживаться во всем тексте программы. Чаще всего для оформления отступов используют табуляцию, устанавливая при этом для нее желаемый шаг. Последняя возможность поддерживается большинством интегрированных сред разработчика. Однако нужно помнить, что оборотная сторона возможности настроить шаг табуляции состоит в том, что в разных текстовых редакторах настройки табуляции могут оказаться различными.

По этой причине в окончательной версии исходного текста рекомендуется каждый символ табуляции заменить на заданное количество символов пробела. Большинство текстовых редакторов предоставляют такую возможность. При расстановке отступов рекомендуется придерживаться следующих общепринятых правил [9]:

1) операторы одного уровня иерархии должны быть размещены с равным отступом (табл. 4.2);

Таблица 4.2

Равный отступ

Строка	Неправильно	Правильно

1.
2.
3.
4.
	`printf("Enter value: ");`
	`  scanf( "%d", &dim );`
	`  ptr = new int [dim];`
	`ptr [0] = 0;`
		`printf("Enter value: ");`
		`scanf( "%d", &dim );`
		`ptr = new int [dim];`
		`ptr [0] = 0;`

2) подчиненные операторы должны быть сдвинуты вправо по отношению к управляющему оператору, образуя следующий уровень иерархии (табл. 4.3);

Таблица 4.3

Сдвиг подчинённых операторов

Строка	Неправильно	Правильно

1.
2.
3.
4.
	`if( f == NULL )`
	`printf( "No file\n");`
	`else`
	`printf( "Start..\n");`
		`if( f == NULL )`
		`  printf( "No file\n");`
		`else`
		`  printf( "Start..\n");`

3) размер сдвига должен быть постоянным (табл. 4.4);

Таблица 4.4

Постоянный размер сдвига подчинённых операторов

Строка	Неправильно	Правильно

1.
2.
3.
4.
	`if( ptr == NULL )`
	`  return -1;`
	`for( i=0; i<dim; i++ )`
	`  ptr [i] = i;`
		`if( ptr == NULL )`
		`  return -1;`
		`for( i=0; i<dim; i++ )`
		`  ptr [i] = i;`

4) «лесенка» должна отражать структурную вложенность языковых конструкций. Приведем примеры реализации лесенки:

```
short Sign( float X )
{
   // выдает знак числа X
   if ( X > 0 ) return 1;
   else
      if ( X < 0 ) return -1;
      else
         return 0;
}
void Equation( float A, float B, float C,
               float* X1, float* X2, char* Num )
{
   // нахождение действительных корней квадратного уравнения;
   // A, B, C -- коэффициенты
   // X1, X2 -- корни (если действительного решения нет, то
   // полагаются равными 0);
   // Num -- число корней (0, 1, или 2)

   float D;

   D = sqr( B ) - 4 * A * C;
   if ( D < 0 )
   {
      *Num = 0;
      *X1 = 0;
      *X2 = 0;
   }
   else
   {
      *X1 = ( -B + sqrt( D ) ) / ( 2 * A );
      *X2 = ( -B - sqrt( D ) ) / ( 2 * A );
      if ( *X1 == *X2 ) *Num = 1;
      else *Num = 2;
   }
}
```

4.3.4. Операторные скобки

Существуют два рекомендуемых стиля расстановки операторных скобок. При использовании первого стиля открывающая фигурная скобка помещается на той же строке, что и управляющая конструкция, а закрывающая – строго на уровне управляющей конструкции:

```
   int factorial( int n ) {
      if( n > 1 )
         return n * factorial( n-1 );
      if( n < 0 ) {
         fprintf( stderr, "Error: negative argument\n" );
         return -1; //Заведомо невозможный результат
      }
      return 1;
   }
```

Второй подход проиллюстрируем на том же примере:

```
int factorial( int n )
{
   if( n > 1 )
      return n * factorial( n-1 );
   if( n < 0 )
   {
      fprintf( stderr, "Error: negative argument\n" );
      return -1; //Заведомо невозможный результат
   }
   return 1;
}
```

Из сравнения двух приведенных примеров видно, что отличие состоит в положении открывающей скобки. Однако закрывающая скобка в обоих случаях должна находиться на уровне управляющего оператора или описания. Некоторые специалисты полагают, что второй подход оправдан в большей мере, т. к. обеспечивает улучшенную наглядность. Открывающая и закрывающая скобки при этом располагаются строго друг под другом, что помогает находить начало и конец составного оператора, функции или описания класса. Однако другие эксперты больше склоняются к первому способу. Программист может выбрать любой из них, но далее необходимо придерживаться выбранного способа на протяжении всего текста программы.

Следует отметить, что наличие или отсутствие скобок не влияет на наличие и размер отступов, что видно в приведенных примерах.

Иногда встречается третий стиль, который похож на второй, но отличается расположением скобок где-то внутри отступа:

```
int factorial( int n )
 {
  if( n > 1 )
    return n * factorial( n-1 );
  if( n < 0 )
   {
    fprintf( stderr, "Error: negative argument\n" );
    return -1; //Заведомо невозможный результат
   }
  return 1;
 }
```

Третий стиль по наглядности уступает второму, поскольку не только не вносит дополнительной ясности, но и несколько снижает наглядность исходного текста. Действительно, выбор положения скобок где-то внутри отступа достаточно произволен и ничем не обоснован, а количество вертикальных зрительных линий отступов из-за этого удваивается и составляет 8–10 линий вместо обычных 4–5, что снижает наглядность без повышения информативности.

4.3.5. Пробелы

Характерная особенность визуального восприятия информации человеком такова, что знаки пробелов распознаются лучше других символов синтаксиса. В связи с этим, отдельные элементы текста необходимо всегда выделять пробелами, даже если обсуждаемые элементы текста, возможно, уже выделены какими-то другими знаками препинания (скобками, запятыми, точками с запятой и т. п.). Принципиально важно всегда выделять пробелами стоящие рядом операторы и списки аргументов функций (табл. 4.5).

Таблица 4.5

Расстановка пробелов

Строка	Неправильно	Правильно
1.	`while(i++<dim)`	`while ( i++ < dim )`
2.	`move(a,b,ptr [i++]);`	`move ( a, b, ptr [ i++ ] ) ;`

Добавочные пробелы могут быть также использованы также для выравнивания сходных по смыслу или однотипных частей выражений для того, чтобы улучшить их наглядность, например, при объявлении переменных или для серии присваиваний:

```
int a, size;
char *buf;
float lenght1, lenght2;
. . .
a       = 1;
lenght1 = GetLength();
lenght2 = 0;
size    = (int) lenght1;
```

4.3.6. Пустые строки

Добавление пустых строк в текст программы является важным средством для разграничения различных участков программы. В следующих случаях рекомендуется добавлять пустые строки:

1) объявления переменных:

```
char str [80];
int counter = 0;

fgets( str, 79, infile);
counter++;
```

2) последовательности однотипных инструкций или директив:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define NAME_SIZE 256
#define MAX_LEN 3000
```

3) функции:

```
int main()
{
. . .
}

char *get_name(FILE *f)
{
. . .
}
```

4) любые логически завершенные блоки кода:

```
printf( "Enter size and delta: " ); //Блок ввода данных
scanf( "\%\,d", \&\,size );
scanf( "\%\,f", \&\,delta );

for( i=0; i<size; i++ ) //Блок использования данных
{
  a [i] -= delta;
  b [i] += delta;
}
```

4.3.7. Улучшение читаемости программ

Приведем рекомендации, повышающие читаемость программ:

1) операнды бинарных операций (+, = и т. п.) рекомендуется всегда отделять от знака операции одним пробелом, например

```
Sum = A + B;
```

2) при перечислении идентификаторов после запятой ',' рекомендуется всегда ставить один пробел, например

```
printf( "Сумма: %d; Разность: %d.", A + B, A - B );
unsigned short Day, Month, Year;
unsigned char i, j, k, l, m, n;
```

3) непосредственно после каждого оператора рекомендуется всегда писать символ-разделитель точка с запятой ';', например

```
switch ( Num )
{
   case 1:  printf( "Один..." ); break;
   case 2:  printf( "Два..." );  break;
   case 3:  printf( "Три..." );  break;
   default: printf( "Много!" );  break;
}
```

4) 16-ричные числа рекомендуется всегда писать прописными буквами, например

```
#define BadDate 0xFFFF
#define kbEnter 0x0D // Код клавиши <Enter>.
```

Следует отметить, что в настоящей главе приведена сравнительно небольшая выборка требований к оформлению исходного текста программного кода. Набор этих требований и глубина обсуждения были продиктованы прежде всего ориентированностью на их использование в учебном процессе в рамках курсового проектирования по микропроцессорной технике. При необходимости руководствоваться более полной и детальной информацией о требованиях к оформлению программного кода следует, в первую очередь, обращаться к Единой системе программной документации (ЕСПД), которая представляет собой комплекс государственных стандартов (ГОСТ) Российской Федерации, устанавливающих взаимосвязанные правила разработки, оформления и обращения программ и программной документации. Кроме того, существует большое количество корпоративных стандартов, отражающих специфические требования различных компаний. Помимо этого существуют многочисленные книги и руководства по оформлению программного кода (например, [10]), которые также могут оказаться полезными. Конкретные сведения о практических аспектах оформления программ можно найти в прил. В и методических указаниях [11].

Алфавитный указатель

М

Н

О

П

С

Т

У

Ф

Я

Список библиографических ссылок

1. Огородников И. Н. Микропроцессорная техника : учебник. Екатеринбург : УГТУ-УПИ, 2007. 380 с.

2. Огородников И. Н. Микропроцессорная техника : практический курс. Екатеринбург : УрФУ, 2012. 137 с.

3. Каспер Э. Программирование на языке ассемблера для микроконтроллеров семейства i8051. М. : Горячая линия-Телеком, 2003. 192 с.

4. Cx51 compiller. Optimizing C compiller and library reference for classic and extended 8051 microcontrollers : user's guide. Keil Software, 2001. 402 p.

5. Учебный стенд SDK-1.1 : руководство пользователя. [Электронный ресурс]. СПб. : ЛМТ, 2006. 100 с. URL: http://lmt.cs.ifmo.ru (дата обращения: 01.01.2020).

6. Спецификация ADuC812. [Электронный ресурс] / пер. с англ. Б. Л. Горшкова, Ю. М. Зайцева, В. И. Силантьева. СПб. : АВТЭКС, 1999. 30 с. URL: http://www.autex.spb.ru (дата обращения: 01.01.2020).

7. ADuC841/ADuC842/ADuC843: MicroConverter, 12-Bit ADCs and DACs with Embedded High Speed 62-kB Flash MCU Data Sheet (Rev 0, 11/2003) [Electronic resource]. Norwood : One Technology Way, 2003. 88 p. URL: http://embedded.ifmo.ru/index.php/support/sdk-11 (date of access: 01.01.2020).

8. Лукичев А. Н. Отличия в программировании SDK-1.1 с ADuC842, ADuC831 и ADuC812. [Электронный ресурс]. СПб. : ЛМТ, 2004. 10 с. URL: http://www.lmt.ifmo.ru (дата обращения: 01.01.2020).

9. Цымблер М. Л. Требования к оформлению программ на языке Turbo Pascal. [Электронный ресурс]. 2010. URL : http://www.mzym.susu.ru/papers (дата обращения: 18.01.2019).

10. McConnell S. Code Complete : A Practical Handbook of Software Construction. 2nd ed. Microsoft Press, 2004. 952 p.

11. Кичигин В. Н., Мясников И. Е., Тимошенко С. И. Оформление курсовых и дипломных проектов : методические указания для студентов технических специальностей. Екатеринбург : УГТУ-УПИ, 2005. 80 с.

Приложение А. Подключение стенда к usb-порту компьютера

Для подключения лабораторных стендов sdk-1.1 и sdk-1.1/s к com-порту компьютера, содержащего инструментальную систему, в качестве штатного средства предусмотрен последовательный интерфейс RS232 в 9-контактном варианте. Однако современные стационарные компьютеры и ноутбуки чаще оборудованы usb-портами. В таком случае для подключения лабораторных стендов к компьютеру необходимо использовать переходник-преобразователь usb—RS232.

Кабель-переходник usb—RS232 типа ML-A-043 (*Ruichi*, Китай) предназначен для подключения устройств, имеющих com-порт (последовательный порт RS232), к usb-порту компьютера. Адаптер переходника может поддерживать скорости обмена до 1 Мб/с. В комплект поставки входит адаптер ML-A-043, снабженный шнуром-удлинителем длиной 70 см, и мини-CD диск с драйверами (рис. А.1). Драйверы можно загрузить также с сайта производителя микросхемы FTDI [1].

Рис. А.1. Кабель-переходник usb–RS232 типа ML-A-043 (*Ruichi*, Китай). Фотография с сайта производителя

В инсталляторе драйвера поддержаны операционные системы: winXP/win7/win10 и Linux. Драйвер переходника образует в операционной системе виртуальный com-порт, с которым и должно работать программное обеспечение подключаемого устройства. Номер виртуального com-порта индивидуален для конкретного компьютера. В операционной системе Windows для определения номера com-порта следует использовать Device Manager.

[1] URL: http://www.ftdichip.com/Drivers/CDM/CDM21228_Setup.zip (дата обращения: 01.01.2020).

Приложение Б. Функции и константы библиотеки sdk_base

Таблица Б.1

Функции библиотеки sdk_base

Название функции	Операция	Назначение функции
getbyte	R	Получение байта из области xdata
getkey	R	Получение байта от клавиатуры
lgetchar	R	Получение байта от ЖК-дисплея
lputchar	W	Отправка байта в ЖК-дисплей
putbyte	W	Отправка байта в область xdata
svdisp	W	Отправка байта на светодиоды
get_idata	R	Получение байта из области idata
get_lpt	R	Получение байта из внешнего порта
put_idata	W	Отправка байта в область idata
put_lpt	W	Отправка байта во внешний порт
get_uart	R	Получение байта данных из uart
put_uart	W	Отправка байта данных в uart
uart_ini	W	Отправка управляющего байта в uart
getack	W,R	Проверка готовности к обмену (пинг)
getmode	R	Получение управляющего байта из часов
gettime	R	Получение блока данных из часов
putmode	W	Отправка управляющего слова в часы
puttime	W	Отправка блока данных в часы
receiveblock	R	Получение текущего времени из часов
sendblock	W	Отправка текущего времени в часы
adc_ini	W	Отправка управляющего байта в АЦП
adc_single	R	Получение двух байтов данных из АЦП
dac_ini	W	Отправка управляющего байта в ЦАП
dac_single	W	Отправка двух байтов данных в ЦАП

Таблица Б.2

Предопределенные константы библиотеки sdk_base

Названия библиотечных констант				
ADC_CON812	ADC_DAT3	DATA_IND	LCD_CR	RTC_PM
ADC_CON842	ADUC812	EXT_LO	LCD_BF	SV
ADC_TSENS	ADUC842	EXT_HI	RTC_AM	LCD_POS
ADC_DAT0	C_IND	ENA	RTC_12	RTC_START
ADC_DAT1	DAC_0	KB	RTC_24	RTC_STOP
ADC_DAT2	DAC_1	LCD_CLR	RTC_HH	

Приложение В. Курсовая работа по микропроцессорной технике

Микропроцессорная техника ныне настолько широко используется практически в любой отрасли народного хозяйства, что современное развитие науки, техники и технологий уже невозможно представить без применения в них микропроцессорной техники. Вполне естественно, что важнейшими критериями оценки эффективности подготовки студентов (бакалавров, специалистов и магистрантов) при этом являются знания, умения, навыки и соответствующие компетенции в области микропроцессорной техники. Значимым промежуточным этапом практического курса изучения микропроцессорной техники является курсовая работа. Подытожены методические указания и рекомендации по выполнению курсовой работы, сформулированы необходимые требования к построению и содержанию курсовой работы, ее оформлению, а также критерии, которыми нужно руководствоваться при выборе темы работы.

Общие положения

Объем и содержание учебного курса «Микропроцессорная техника», составной частью которого является курсовая работа, определяются Федеральными государственными стандартами по направлениям «Ядерные физика и технологии», «Биотехнические системы и технологии», специальности «Электроника и автоматика физических установок», а также учебными планами соответствующих направлений специальностей подготовки бакалавров, магистров и специалистов.

Приведенные методические указания предназначены в первую очередь для студентов направления «Ядерные физика и технологии», которые в осеннем семестре выполняют курсовую работу по дисциплине «Микропроцессорная техника». В то же время, разработанные методические указания и рекомендации могут оказаться весьма полезными также студентам всех упомянутых выше направлений и специальностей при выполнении курсовых работ и проектов по смежным дисциплинам, в которых используются элементы микропроцессорной техники.

Этапу курсового проектирования обязательно должно предшествовать предварительное аудиторное и самостоятельное изучение базового курса «Микропроцессорная техника», а также дисциплин «Цифровая электроника», «Аналоговая схемотехника», «Физические основы электронной техники» и «Общие основы электротехники».

Цель и задачи курсовой работы

Курсовая работа определяется как учебный документ, который должен быть подготовлен и защищен каждым студентом на заданном промежуточном этапе обучения в соответствии с учебным планом. Непосредственное выполнение курсовой работы классифицируется как один из видов самостоятельной работы студентов, выполняемой в рамках бюджета времени, отводимого на изучение дисциплины.

Целью курсовой работы является систематизация и закрепление теоретических и практических знаний студентов по отдельным дисциплинам путем практического применения этих знаний при проектировании микропроцессорных контрольно-измерительных и управляющих систем, а также систем сбора и обработки цифровой информации.

В рамках курсовой работы перед каждым студентом ставятся следующие задачи:

1) развить навыки самостоятельной работы, проверить свои способности самостоятельно, творчески применять на практике теоретические знания, полученные ранее при изучении базовой дисциплины «Микропроцессорная техника»;

2) освоить современные технологии проектных и научных работ;

3) обучиться методам выбора и обоснования технических решений, получить соответствующие навыки и умения в проектировании контрольно-измерительных и управляющих устройств на однокристальных микроконтроллерах;

4) изучить современные стандарты, а также типовые системы автоматизированного проектирования и подготовки проектно-конструкторской документации.

Помимо сформулированных выше задач, в процессе курсового проектирования студенты должны приобрести практические навыки составления алгоритмов заданных операций и доведения поставленной задачи до конкретного программно-аппаратного технологического решения. Для достижения цели и выполнения поставленных задач студентам необходимо не только хорошо освоить теоретический материал, но и проявить творческую инициативу в подборе и изучении специальной литературы, в т. ч. на иностранном языке.

Тематика курсовых работ

Курсовая работа считается одним из видов самостоятельной работы студента, поэтому к ее выполнению рекомендуется относиться с особым вниманием и тщательностью. В рамках курсовой работы каждый студент получает индивидуальное задание, которое выдается руководителем и предусматривает техническое решение какой-либо

специализированной технологической задачи путем применения микропроцессорных средств цифровой обработки информации или управления объектами в режиме реального времени.

Тематика курсовой работы как правило связана с проектированием заданных контроллерных устройств, выполняющих разнообразные контрольно-измерительные и управляющие функции, а также функции по обработке данных, которые поступают в режиме реального времени. В некоторых случаях тема курсовой работы может быть связана с научными или производственными интересами студента, лежащими в русле изучаемой дисциплины.

Для выполнения курсовой работы студенту предоставляются исходные данные, которые задаются в таком виде, чтобы стимулировать поиск прогрессивных схемных и алгоритмических решений и получение высоких целевых показателей при обработке потоков данных. Основной акцент при разработке содержания отчета о выполнении курсовой работы должен делаться на получение студентом в процессе проектирования практических навыков в выборе алгоритмов решения типовых задач, возникающих в реальной практике перед разработчиком аппаратно-программных средств.

Руководитель разрабатывает тематику курсовой работы в соответствии с учебным планом дисциплины «Микропроцессорная техника» и специализацией студентов.

Кроме вариантов индивидуальных заданий, предлагаемых руководителем, возможен выбор темы работы, которая может быть связана с госбюджетными или хоздоговорными научно-исследовательскими и опытно-конструкторскими работами, выполняемыми на кафедре, а также с заказами предприятий и организаций соответствующего профиля. Активные участники учебно-исследовательской работы студентов (УИРС) могут предложить свою собственную тему. Если эта тема соответствует программе курса, то может быть утверждена руководителем в качестве индивидуального задания на курсовую работу.

В зависимости от объема решаемых задач темы курсовых работ могут быть как индивидуальными, рассчитанными на выполнение одним студентом, так и комплексными. Выполнение комплексной темы подразумевает привлечение нескольких студентов, каждому из которых отводится самостоятельная часть из общей работы. Предпочтение следует отдавать комплексным темам, которые позволяют наиболее полно решить поставленную задачу и достигнуть завершающей стадии.

Помимо выдачи студентам индивидуальных заданий, на кафедре оформляется официальное распоряжение об утверждении тем курсовых работ, подписанное заведующим кафедрой. Копия этого распоряжения размещается на доске объявлений кафедры.

Порядок выполнения курсовой работы

Получив индивидуальное задание студент должен сразу же приступить к выполнению курсовой работы, не дожидаясь, когда рассматриваемый теоретический или практический вопрос будет рассмотрен в лекционном курсе или лабораторном практикуме. Чтобы качественно и своевременного выполнить курсовую работу необходимо придерживаться следующей рекомендуемой последовательности действий:

1) сразу же ознакомиться с выданной преподавателем темой, уяснить ее и провести тщательный анализ текста индивидуального задания на курсовую работу. При необходимости рекомендуется уточнить у руководителя аспекты работы, которые вызывают вопросы;

2) самостоятельно изучить полученную тему по современным литературным и интернет-источникам. Установить функциональное назначение проектируемой системы;

3) составить предварительный план предстоящей работы и, детально обсудив его с руководителем, утвердить подготовленный план;

4) ознакомиться со специальной технической литературой по теме курсовой работы и изучить поставленный вопрос. При выполнении курсовой работы особая роль отводится работе студента с технической литературой, в т. ч. на иностранном языке;

5) с учетом изученной специальной литературы составить эскизный проект функциональной схемы аппаратной части, провести обоснование и выбор структурной схемы микропроцессорной системы, интегральных схем и других электронных компонентов;

6) составить и обосновать блок-схему алгоритма работы контроллера и приступить к ее описанию;

7) в соответствии с блок-схемой алгоритма и принципами структурного программирования начать проектировать базовые модули программы. На данном этапе рекомендуется использовать интегрированную среду разработки Keil µVision: макроассемблер A51, которые уже знакомы студентам по лабораторному практикуму, а также C51 и библиотеку `sdk_base.lib` для доступа к функциональным элементам лабораторного стенда `sdk-1.1` и `sdk-1.1/s`;

8) настоятельно рекомендуется тщательно отлаживать и проверять каждый функционально законченный блок программы задолго до его фактического размещения в общем программном модуле;

9) необходимо добиться работоспособности всей программы, убедиться в ее соответствии всем требованиям индивидуального задания. Продемонстрировать работу программы преподавателю и зафиксировать результаты испытания. Для доказательства правильности функционирования разработанного контроллера может потребоваться подключение дополнительного лабораторного оборудования, имеющегося

в лаборатории микропроцессорной техники. Для верификации программы может потребоваться разработка тестовых примеров, наборов тестовых данных или дополнительных тестовых программ;

10) после успешного испытания и одобрения руководителем можно приступить к оформлению отчета о курсовой работе.

Для курсовых работ, которые выполняются по заданию кафедры и связаны с разработкой и изготовлением макетов и лабораторных образцов, по согласованию с руководителем допускается оформление только отчета о курсовой работе объемом 10. . . 15 страниц с включением в нее в качестве приложений всех необходимых структурных, принципиальных или аналогичных им по смыслу схем.

Для курсовых работ, направленных на решение самостоятельных задач в рамках госбюджетных или хоздоговорных работ, по согласованию с руководителем допускается защита без оформления текстовых и графических материалов в случае, если эти материалы включены в отчеты по соответствующей научно-исследовательской или опытно-конструкторской работе.

Требования к структуре и оформлению работы

В состав курсовых работ (проектов) входят текстовые и графические документы, а также может входить программная и технологическая документация. Правила оформления курсовых работ (проектов) есть в стандарте предприятия СТП УПИ 1–96. Стандарт предприятия опирается на государственные стандарты Российской Федерации. На текущий момент времени это основные стандарты: ГОСТ 2.105–95, ГОСТ 6.30–97 и ГОСТ 7.32–2001. Конкретизация специфики оформления учебной документации по профилю обучения содержится в методических рекомендациях по оформлению курсовых и дипломных работ [11]. Методические рекомендации не подменяют действующих государственных стандартов Российской Федерации, которые обязательны для изучения и играют главенствующую роль.

Законченная курсовая работа состоит из отчета о выполнении курсовой работы и графического материала, которые должны в совокупности давать достаточно полное представление о разрабатываемом устройстве, основных принципах его работы, о решениях, положенных в основу разработки функциональных схем и т. д.

Студент несет полную ответственность за точность и правильность вычислений, работоспособность программного кода, качество оформления отчета о выполнении курсовой работы и графических материалов, за своевременное выполнение и представление работы к защите.

Хорошо выполненные курсовые работы могут быть рекомендованы для участия в конкурсах или студенческих конференциях.

Ниже приводится более подробное описание основных разделов отчета о курсовой работе.

Отчет о курсовой работе является основным документом, представляемым к защите. Качество его оценивается полнотой, точностью и логикой изложения, а также аккуратностью и правильностью заполнения составляющих частей: текста, рисунков, таблиц и библиографического списка.

Объем отчета должен составлять примерно 20. . . 30 страниц стандартного формата А4 (210×297 мм), включая схемы и иллюстрации. В отдельных случаях допускается использование большего объема записки и графики большего формата. По краям листа оставляются поля: слева – 25 мм, справа – 10 мм, сверху и снизу – по 20 мм.

Все страницы отчета, в т. ч. с графиками, рисунками, листингом программы и т. п., нумеруются и переплетаются в единую папку – учебный документ.

Схемы, диаграммы, рисунки и другие иллюстративные материалы выполняются на отдельных листах миллиметровой бумаги единообразно по всей работе, с указанием номера рисунка и подрисуночной подписью; все формулы также следует пронумеровать.

Отчет о курсовой работе в общем случае может состоять из нижеследующих составных частей:

1) титульный лист, на котором указывается наименование структурного образования (Министерство науки и высшего образования Российской Федерации), полное наименование университета, института и кафедры, а также тема курсовой работы, ФИО студента и номер группы, ФИО преподавателя, город и год (1 страница). Пример титульного листа приведен в прил. Г.

2) бланк задания на курсовую работу, подписанный преподавателем и утвержденный заведующим кафедрой. Задание на выполнение курсовой работы является нормативным документом, устанавливающим границы и глубину исследования (разработки) темы, а также сроки представления работы на кафедру в завершенном виде. Бланк задания имеет стандартную форму и содержит данные о студенте, название темы работы и краткие исходные данные. Здесь кратко приводится основное содержание отчета о курсовой работе и графического материала. Задание с указанием даты его выдачи подписывается преподавателем и утверждается заведующим кафедрой. Задание на курсовое проектирование необходимо разместить в начале пояснительной записки сразу после титульного листа (1 страница).

После выполнения курсовой работы руководитель делает отметку на бланке о ее завершении, а по окончании защиты комиссия проставляет оценку. Без правильно заполненного бланка задания и отметки о ее завершении курсовая работа к защите не допускается. Форма бланка заданий на курсовую работу приведена в прил. Г;

3) оглавление, которое включает название разделов и номера страниц, соответствующих началу каждого из них (1 страница);

4) перечень всех использованных сокращений и обозначений, содержащий составленный в алфавитном порядке список сокращений и аббревиатур, использованных в тексте и на чертежах, каждое из которых должно быть расшифровано (1 страница);

5) введение, содержащее краткую вводную информацию, касающуюся предметной области курсовой работы (1. . . 2 страницы);

6) инженерная формулировка проектного задания и техническое обоснование выбранных решений со ссылками на литературные источники. Проектное задание – это полное и точное определением всех требований к разрабатываемому устройству. Обязательно приводится связь проектируемого устройства с внешними для него источниками потоков данных, т. е. интерфейс проектируемого устройства. Техническое обоснование выбранного решения представляет собой аналитический обзор технической литературы по теме курсовой работы, включающий периодические издания и интернет-источники. Основное назначение этого обзора – показать преимущества и недостатки выбранных технических решений в сравнении с известными альтернативными решениями. При необходимости рассматриваются также теоретические аспекты вопроса (2. . . 3 страницы).

7) разработка и описание функциональной схемы, выбор и обоснование технологических параметров, необходимых для последующего программирования (адреса устройств, векторы прерываний, режимы функционирования встроенных узлов, управляющие слова, флаги, сигналы квитирования и т. п.). В этом разделе необходимо описать принципы построения функциональной схемы контроллера, взаимодействие основных его устройств при вводе, обработке и выводе информации, способы формирования управляющих сигналов и организации с их помощью управления, порядок программирования и перепрограммирования больших интегральных схем (БИС), а также выполнить расчет отдельных его элементов. При необходимости привести временные диаграммы работы элементов контроллера либо таблицы их состояний. На чертежах все выводы БИС обозначить в соответствии с общепринятой методикой их обозначения, взятой из ЕСКД и стандартов на микросхемы.

Проработка этого этапа необходима даже в том случае, когда в качестве прототипа выбирается стандартное аппаратное решение, например стенд SDK-1 (5. . . 7 страниц);

8) разработка, графическое представление и описание блок-схемы алгоритма программы. Алгоритм работы устройства – один из основных документов, к его разработке следует отнестись особенно внимательно, поскольку при неверно выбранном алгоритме теряют смысл самые изысканные технические решения (3. . . 4 страницы);

9) разработка программы на языках ассемблера или Keil C51 и ре-

зультаты компиляции. Описание программы и использованных библиотечных процедур (2. . . 3 страницы). Листинг программы с подробными комментариями поместить в приложение;

10) результаты испытаний, содержательные выводы и рекомендации по дальнейшему развитию работы (1. . . 2 страницы);

11) список использованных литературных и интернет-источников (1 страница);

12) приложение, содержащее графический материал и листинг программы (количество страниц – по необходимости).

В процессе выполнения перечисленных этапов работы формируются текстовый и графический материалы, которые в дальнейшем составят основу отчета о курсовой работе. Состав разделов отчета о выполнении курсовой работы для конкретного задания уточняется во время консультации с преподавателем.

Графическая часть курсовой работы выполняется обычно на листах формата А3. Допускается оформление графических материалов на масштабно-координатной (миллиметровой) бумаге – на лицевой или изнаночной стороне – карандашом, но с обязательным использованием необходимых чертежных инструментов. Условные обозначения, шрифты и масштабы чертежа должны соответствовать требованиям Единой системы конструкторской документации ЕСКД. Допускается использование специализированных компьютерных пакетов разработки и оформления конструкторской документации, например, КОМПАС или Autocad.

Рекомендуются следующие минимальные нормы представления графических материалов:

1) функциональная схема контроллера – 1 лист;

2) блок-схема алгоритма программы – 1 лист.

Перечень графического материала задан в объеме, предусматривающем обучение студента оформлению технической документации. Конкретное содержание графической части уточняется при выдаче задания. В случае необходимости по согласованию с руководителем допускается изменение норм представления графического материала, внесение части из них в расчетно-пояснительную записку и т. п. при непременном сохранении общего объема не менее двух листов.

Контроль за выполнением работы

Руководитель контролирует все этапы выполнения курсовой работы студентом. Кроме того, руководитель:

1) проводит консультации и обсуждения со студентом правильности принимаемых решений;

2) контролирует соблюдение плановых сроков выполнения отдельных этапов;

3) проверяет все материалы, составляющие работу, на предмет их готовности к защите.

За содержание, форму работы и ее качество ответственность несет исполнитель – студент.

Устанавливаются три контролируемых рубежа выполнения курсовой работы. К первому рубежному контролю должны быть: сформулировано проектное задание, выполнен аналитический обзор литературы по теме, разработан эскиз функциональной схемы (т. е. 25. . . 30 % от общего объема курсовой работы).

Ко второму рубежному контролю следует представить полностью проработанную функциональную схему и блок-схему ее функционирования (т. е. должно быть готово 60 % общего объема).

К третьему рубежному контролю работа должна быть закончена. Объекты проектирования следует испытать в лаборатории микропроцессорной техники и продемонстрировать их работу руководителю. После одобрения руководителя вся документация должна быть оформлена и предъявлена преподавателю на проверку. Готовность курсовой работы определяется руководителем и подтверждается его подписью на расчетно-пояснительной записке и графическом материале. Курсовая работа считается готовой, если выполнены все пункты, предусмотренные заданием. Работы, не прошедшие лабораторные испытания, в исключительных случаях по решению заведующего кафедрой могут быть также допущены к защите. Однако такие работы, как правило, не могут претендовать на получение высших оценок.

Все материалы курсовой работы после установления ее готовности представляются комиссии, назначаемой кафедрой, и защищаются студентом по всем разделам, предусмотренным заданием. По результатам выполнения работы и защиты выставляется оценка с учетом:

1) объема и качества выполнения работы, оригинальности и самостоятельности решений;

2) знаний по вопросам, связанным с разработкой функциональной схемы контроллера, блок-схемы алгоритма его функционирования и программы на языке ассемблера и Keil C51;

3) умения излагать результаты работы, обосновывать и защищать принятые решения и отвечать на заданные вопросы.

Полностью подготовленная и оформленная курсовая работа сдается на проверку преподавателю. При наличии замечаний в рецензии на работу студент должен дать на них письменный ответ, а в случае необходимости внести в работу исправления. После доработки исправлений курсовая работа допускается к защите. Качество доработки оценивает преподаватель, который на титульном листе ставит визу о допуске работы к защите.

На защиту студент представляет отчет о выполнении курсовой работы, подписанный преподавателем и рецензию на курсовую работу, подписанную специалистом. Пример бланка рецензии приведен

в прил. Г. Студент должен продемонстрировать, что указанные выше документы (отчет и рецензия) размещены в его электронном портфолио. При отсутствии перечисленных документов (отчет и рецензия) и их электронных копий в портфолио студент к защите не допускается.

Защита работы производится индивидуально каждым студентом на заседании комиссии, назначаемой распоряжением заведующего кафедрой. На изложение основного содержания курсовой работы студенту дается около 7 мин, доклад иллюстрируется графическим материалом. Оценка работы – дифференцированный зачет (зачет с оценкой).

В процессе защиты студенту могут быть заданы разнообразные вопросы по содержанию работы, по теоретической ее части, по алгоритму и функциональным схемам отдельных узлов.

Комиссия оценивает работу по ее качеству и уровню защиты и проставляет оценку на титульном листе работы и в зачетной книжке студента. При получении неудовлетворительной оценки студент защищает курсовую работу повторно. Повторная защита с целью получения повышенной оценки, как правило, не допускается. В исключительных случаях по разрешению заведующего кафедрой студент может получить новое задание и выполнить курсовую работу заново с последующей ее защитой.

По завершении защиты отчет о выполнении курсовой работы, рецензия и графические материалы остаются для хранения на кафедре.

Приложение Г. Документы для выполнения курсовой работы

Титульный лист отчета о курсовой работе

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина» (УрФУ)
Физико-технологический институт
Кафедра экспериментальной физики

Оценка _____
Руководитель курсового
проектирования _____

Члены комиссии _____
_____

Дата защиты _____

ОТЧЕТ
о курсовой работе

Студент: _____ _____

Группа: _____

Екатеринбург
20__

Бланк задания на курсовую работу

Уральский федеральный университет имени первого Президента России Б.Н.Ельцина

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Уральский федеральный университет имени первого Президента России Б. Н. Ельцина» (УрФУ)
Физико-технологический институт
Кафедра экспериментальной физики

УТВЕРЖДАЮ:

Завкафедрой_____ В. Ю. Иванов
« 18 » Ноября 2018 г.

Дисциплина: Микропроцессорная техника

Задание на курсовую работу

Студент: Сергеев Иван Сергеевич.
Группа: Фт-450013.
Направление подготовки: 14.03.02 – Ядерные физика и технологии.

1. Тема курсовой работы: «Перекодировка функции».

2. Содержание работы, в том числе состав графических работ и расчётов:
Листинг программы на языке ассемблера с комментариями; блок-схема алгоритма программы; блок-схема аппаратной базы; результаты испытаний.

3. Дополнительные сведения.
Модифицировать подпрограмму `getkey`, используя возможности битового процессора микроконтроллера: составить подпрограмму для преобразования четырехбитного кода нажатой клавиши (выходной код до модификации программы) в четырехбитный код, в точности соответствующий двоичному представлению изображенного на клавише символа (`0...F`). Платформа `x51`, микроконтроллер `ADUC842`, ассемблер `A51`. Использовать не табличное преобразование кода, а синтез рассчитанной переключательной функции.

4. План выполнения курсовой работы:

Наименование элементов проектной работы	Сроки	Примечания	Отметка о выполнении
Изучение стенда `SDK`	29.10–10.11		
Разработка программы	12.11–24.11		
Испытание программы	26.11–01.12		
Оформление результатов	03.12–22.12		

Руководитель _____ /И. Н. Огородников/

Бланк рецензии на курсовую работу

Министерство науки и высшего образования Российской Федерации
ФГАОУ ВО «Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина»

РЕЦЕНЗИЯ
на курсовую работу

Студента _____ группы _____
Тема курсовой работы _____

Модуль/дисциплина _____
1. Соответствие результатов выполнения работы целям и задачам курсового проектирования результатам обучения по дисциплине/модулю

2. Оригинальность и самостоятельность выполнения работы _____

3. Полнота и глубина проработки разделов _____

4. Общая грамотность и качество оформления текстового документа и графических материалов _____

5. Вопросы и замечания _____

6. Общая оценка работы _____

Сведения о рецензенте:

Ф. И. О. _____
Должность _____
Место работы _____
Ученое звание _____ Ученая степень _____

Подпись _____ Дата _____

Учебное издание

Огородников Игорь Николаевич

МИКРОПРОЦЕССОРНАЯ ТЕХНИКА. ВВЕДЕНИЕ В KEIL C51

Редактор *К. А. Поташев*
Верстка *И. Н. Огородникова*

Подписано в печать 09.07.2021. Формат 70×100 1/16.
Бумага писчая. Цифровая печать. Усл. печ. л. 8,1.
Гарнитура Antiqua. Уч.-изд. л. 5,1. Тираж 30 экз. Заказ 148.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8 (343) 375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: 8 (343) 358-93-06, 350-58-20, 350-90-13
Факс: 8 (343) 358-93-06
http://print.urfu.ru

9 785799 633011

ОГОРОДНИКОВ ИГОРЬ НИКОЛАЕВИЧ

Доктор физико-математических наук, профессор кафедры экспериментальной физики Физико-технологического института УрФУ. Более 35 лет занимается теоретическими и экспериментальными исследованиями динамики электронных возбуждений, локализованных состояний, дефектов, люминесценции и термостимулированных процессов в широкозонных диэлектриках — оптических материалах для современной коротковолновой лазерной техники, нелинейной и интегральной оптики.

Автор более 450 научных публикаций, 6 изобретений, 24 учебно-методических изданий, в том числе 6 учебных пособий и 1 учебника. Подготовил 11 кандидатов и 1 доктора наук.

Член редакционной коллегии международного научного журнала *Radiation Measurements* (*Elsevier*), зарегистрирован в Федеральном реестре экспертов научно-технической сферы, награжден нагрудным знаком «Почетный работник высшего профессионального образования».